目錄

Essential Java.《Java 编程要点》



Essential Java, is a book about the Essentials of Java Programming.

There is also a GitBook version of this book: http://waylau.gitbooks.io/essential-java.

《Java 编程要点》是一本 Java 的开源学习教程，主要介绍 Java 中应用广泛的部分（言外之意，本书不涉 Applet 以及 GUI 框架）。本书包括最新版本 Java 8 中的新特性，以及部分 JDK 9里面的内容，图文并茂，并通过大量实例带你走近 Java 的世界！

本书业余时间所著，水平有限、时间紧张，难免疏漏，欢迎指正，

Get start 如何开始阅读

选择下面入口之一：

- https://github.com/waylau/essential-java 的 SUMMARY.md
- http://waylau.gitbooks.io/essential-java 的 Read 按钮

Code 源码

书中所有示例源码，移步至https://github.com/waylau/essential-java的 `samples` 目录下,代码遵循《Java 编码规范》

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/essential-java/issues

Contact 联系作者:

- Blog: waylau.com
- Gmail: waylau521(at)gmail.com
- Weibo: waylau521
- Twitter: waylau521
- Github : waylau

快速开始

本章介绍了如何下载、安装、配置和调试 JDK。

下载、安装 JDK

JDK(Java Development Kit)是用于 Java 开发的工具箱。

在http://www.oracle.com/technetwork/java/javase/downloads/index.html下载

JDK 支持如下操作系统的安装：

操作系统类型	文件大小	文件
Linux x86	154.67 MB	jdk-8u66-linux-i586.rpm
Linux x86	174.83 MB	jdk-8u66-linux-i586.tar.gz
Linux x64	152.69 MB	jdk-8u66-linux-x64.rpm
Linux x64	172.89 MB	jdk-8u66-linux-x64.tar.gz
Mac OS X x64	227.12 MB	jdk-8u66-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	139.65 MB	jdk-8u66-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	99.05 MB	jdk-8u66-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	140 MB	jdk-8u66-solaris-x64.tar.Z
Solaris x64	96.2 MB	jdk-8u66-solaris-x64.tar.gz
Windows x86	181.33 MB	jdk-8u66-windows-i586.exe
Windows x64	186.65 MB	jdk-8u66-windows-x64.exe

安装路径默认安装在 `C:\Program Files\Java\jdk1.8.0_66` 或者 `usr/local/java/jdk1.8.0_66`

注：本书中所使用JDK版本为：Java Platform (JDK) 8u66。 本书所使用的操作系统为：Win7 Sp1 x64。本书的示例是在 Eclipse Mars.1 Release (4.5.1) 工具下编写。

基于 RPM 的 Linux

（1）下载安装文件

文件名类似于 `jdk-8uversion-linux-x64.rpm` 。

（2）切换到 root 用户身份

（3）检查当前的安装情况。卸载老版本的 JDK

检查当前的安装情况，比如：

```
$ rpm -qa | grep  jdk
jdk1.8.0_102-1.8.0_102-fcs.x86_64
```

若有老版本 JDK，则需先卸载老版本：

```
$ rpm -e package_name
```

比如：

```
$ rpm -e jdk1.8.0_102-1.8.0_102-fcs.x86_64
```

（4）安装

```
$ rpm -ivh jdk-8uversion-linux-x64.rpm
```

比如：

```
$ rpm -ivh jdk-8u102-linux-x64.rpm
Preparing...                ########################################### [100%]
   1:jdk1.8.0_102           ########################################### [100%]
Unpacking JAR files...
    tools.jar...
    plugin.jar...
    javaws.jar...
    deploy.jar...
    rt.jar...
    jsse.jar...
    charsets.jar...
    localedata.jar...
```

（5）升级

```
$ rpm -Uvh jdk-8uversion-linux-x64.rpm
```

安装完成后，可以删除 `.rpm` 文件，以节省空间。 安装完后，无需重启主机，即可使用 JDK。

设置执行路径

Windows

增加一个 `JAVA_HOME` 环境变量，值是 JDK 的安装目录。如 `C:\Program Files\Java\jdk1.8.0_66` ，注意后边不带分号

在 `PATH` 的环境变量里面增加 `%JAVA_HOME%\bin;`

在 `CLASSPATH` 增加 `.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;` （前面有点号和分号，后边结尾也有分号。 或者可以写成 `.;%JAVA_HOME%\lib` 如图所示，一样的效果。

UNIX

包括 Linux、Mac OS X 和 Solaris 环境下，在 `~/.profile` 、 `~/.bashrc` 或 `~/.bash_profile` 文件末尾添加：

```
export JAVA_HOME=/usr/java/jdk1.8.0_66
export PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

其中：

- JAVA_HOME 是 JDK 安装目录
- Linux 下用冒号“:”来分隔路径
- $PATH 、$CLASSPATH、 $JAVA_HOME 是用来引用原来的环境变量的值
- export 是把这三个变量导出为全局变量

比如，在 CentOS 下，需编辑 `/etc/profile` 文件。

测试

测试安装是否正确，可以在 shell 窗口，键入：

```
$ java -version
```

若能看到如下信息，则说明 JDK 安装成功：

```
java version "1.8.0_66"
Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.66-b17, mixed mode)
```

最好再执行下 `javac` ，以测试环境变量是否设置正确：

```
$ javac
用法: javac <options> <source files>
其中, 可能的选项包括:
  -g                         生成所有调试信息
  -g:none                    不生成任何调试信息
  -g:{lines,vars,source}     只生成某些调试信息
  -nowarn                    不生成任何警告
  -verbose                   输出有关编译器正在执行的操作的消息
  -deprecation               输出使用已过时的 API 的源位置
  -classpath <路径>            指定查找用户类文件和注释处理程序的位置
  -cp <路径>                   指定查找用户类文件和注释处理程序的位置
  -sourcepath <路径>           指定查找输入源文件的位置
  -bootclasspath <路径>        覆盖引导类文件的位置
  -extdirs <目录>              覆盖所安装扩展的位置
  -endorseddirs <目录>         覆盖签名的标准路径的位置
  -proc:{none,only}          控制是否执行注释处理和/或编译。
  -processor <class1>[,<class2>,<class3>...] 要运行的注释处理程序的名称; 绕过默认的搜索进程
  -processorpath <路径>        指定查找注释处理程序的位置
  -parameters                生成元数据以用于方法参数的反射
  -d <目录>                    指定放置生成的类文件的位置
  -s <目录>                    指定放置生成的源文件的位置
  -h <目录>                    指定放置生成的本机标头文件的位置
  -implicit:{none,class}     指定是否为隐式引用文件生成类文件
  -encoding <编码>             指定源文件使用的字符编码
  -source <发行版>              提供与指定发行版的源兼容性
  -target <发行版>              生成特定 VM 版本的类文件
  -profile <配置文件>            请确保使用的 API 在指定的配置文件中可用
  -version                   版本信息
  -help                      输出标准选项的提要
  -A关键字[=值]                  传递给注释处理程序的选项
  -X                         输出非标准选项的提要
  -J<标记>                     直接将 <标记> 传递给运行时系统
  -Werror                    出现警告时终止编译
  @<文件名>                     从文件读取选项和文件名
```

有读者反映有时候 `java -version` 能够执行成功，但 `javac` 命令不成功的情况，一般是环境变量配置问题，请参阅上面“设置执行路径”章节内容，再仔细检测环境变量的配置。

更多安装细节，可以参考 http://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html，以及 http://docs.oracle.com/javase/tutorial/essential/environment/paths.html

Java 概述

Java 编程语言是一种通用的、并行的、基于类的、面向对象的语言。它被设计得非常简单，这样程序员可以在该语言上流畅的交流。Java 编程语言与 C 和 C++ 有关联，但组织却截然不同，其中也省略了其他语言的一些用法，比如指针。它的目的是作为一个生产性语言，而不是一个研究性语言，因此，在设计上避免了包括新的和未经考验的功能。

Java 编程语言是强类型和静态类型，可以在编译时检测到编译时错误。编译时间通常由翻译程序到与机器无关的字节码表示的。运行时的活动包括加载和执行程序，可选机器代码生成和程序的动态优化所需的类的交联，和实际执行程序。

Java 编程语言是一个比较高层次的语言，在机器表示的细节是无法通过该语言的。它包括自动存储管理，通常使用垃圾收集器，以避免明确释放的安全问题（就像 C 语言的 free 或 C++ 的 delete）。高性能垃圾回收的实现可具有有界的停顿，以支持系统的编程和实时应用。语言不包括任何不安全的结构，如没有索引检查的数组访问，因为这种不安全的结构会导致不可预知的程序行为。

其他让你选择 Java 的理由还包括：

- 丰富的开发工具：有很多开发工具可以让你快速开始 Java 编程之旅，比如 Eclipse、IntelliJ 和 Netbeans
- 庞大的社区： 在世界各地估计有超过 9 百万的 Java 开发人员。这个数字意味着 Java 开发者可选的能够提高自己能力的讨论会、书籍、在线资源、论坛及培训项目的数量是巨大的。在最新的 TIOBE 编程语言排行榜中， Java 已经晋升榜首
- 快速发展的潜质：Oracle 在 Java 8 中引入 Lambda 表达式和 Streams。以及即将到来的 Java 9，我们预测 Java 的实用性将继续增加。
- 对于移动平台的支持：Android 的火爆很大一部分原因是因为 Java。在 Android 上，ART 为高负荷计算提供了接近本地应用的性能。在 iOS 上，RoboVM 使用 LLVM，其使用的是与 C/C++/Objective-C/Swift 相同的后端，提供了比 Objective-C 和 Swift 只高不低的性能。

语言起源

Java平台和语言最开始只是SUN公司在1990年12月开始研究的一个内部项目。SUN公司的一个叫做帕特里克·诺顿的工程师被自己开发的C和C语言编译器搞得焦头烂额，因为其中的API极其难用。帕特里克决定改用NeXT，同时他也获得了研究公司的一个叫做“Stealth计划”的项目的机会。

“Stealth计划”后来改名为“Green计划”，JGosling（詹姆斯·高斯林）和麦克·舍林丹也加入了帕特里克的工作小组。他们和其他几个工程师一起在加利福尼亚州门罗帕克市沙丘路的一个小工作室里面研究开发新技术，瞄准下一代智能家电（如微波炉）的程序设计，SUN公司预料未来科技将在家用电器领域大显身手。团队最初考虑使用C语言，但是很多成员包括SUN的首席科学家比尔·乔伊，发现C和可用的API在某些方面存在很大问题。

工作小组使用的是内嵌类型平台，可以用的资源极其有限。很多成员发现C太复杂以至很多开发者经常错误使用。他们发现C缺少垃圾回收系统，还有可移植的安全性、分布程序设计、和多线程功能。最后，他们想要一种易于移植到各种设备上的平台。　　根据可用的资金，比尔·乔伊决定开发一种集C语言和Mesa语言搭成的新语言，在一份报告上，乔伊把它叫做“未来”，他提议SUN公司的工程师应该在C的基础上，开发一种面向对象的环境。最初，高斯林试图修改和扩展C的功能，他自己称这种新语言为C--，但是后来他放弃了。他将要创造出一种全新的语言，被他命名为“Oak”（橡树），以他的办公室外的树而命名。　　就像很多开发新技术的秘密的工程一样，工作小组没日没夜地工作到了1992年的夏天，他们能够演示新平台的一部分了，包括Green操作系统，Oak的程序设计语言，类库，和其硬件。最初的尝试是面向一种类PDA设备，被命名为Star7，这种设备有鲜艳的图形界面和被称为“Duke”的智能代理来帮助用户。1992年12月3日，这台设备进行了展示。　　同年11月，Green计划被转化成了“FirstPerson有限公司”，一个SUN公司的全资子公司，团队也被重新安排到了帕洛阿尔托。FirstPerson团队对建造一种高度互动的设备感兴趣，当时代华纳发布了一个关于电视机顶盒的征求提议书时（Requestforproposal），FirstPerson改变了他们的目标，作为对征求意见书的响应，提出了一个机顶盒平台的提议。但是有线电视业界觉得FirstPerson的平台给予用户过多地控制权，因此FirstPerson的投标败给了SGI。与3DO公司的另外一笔关于机顶盒的交易也没有成功，由于他们的平台不能在电视工业产生任何效益，公司再并回SUN公司。

1994年6、7月间，在经历了一场历时三天的头脑风暴的讨论之后，约翰·盖吉、詹姆斯·高斯林、比尔·乔伊、帕特里克·诺顿、韦恩·罗斯因和埃里克·斯库米，团队决定再一次改变了努力的目标，这次他们决定将该技术应用于万维网。他们认为随着Mosaic浏览器的到来，因特网正在向同样的高度互动的远景演变，而这一远景正是他们在有线电视网中看到的。作为原型，帕特里克·诺顿写了一个小型万维网浏览器，WebRunner，后来改名为HotJava。同年，Oak改名为Java。商标搜索显示，Oak已被一家显卡制造商注册，因此团队找到了一个新名字。这个名字是在很多成员常去的本地咖啡馆中杜撰出来的。名字是不是首字母缩写还不清楚，很大程度上来说不是。虽然有人声称是开发人员名字的组合：JamesGosling（詹姆斯·高斯林）ArthurVanHoff（阿瑟·凡·霍夫）AndyBechtolsheim（安迪·贝克托克姆），或“JustAnotherVagueAcronym”（只是另外一个含糊的缩写）。还有一种比较可信的说法是这个名字是出于对咖啡的喜爱，所以以Java咖啡来命名。类文件的前四个字节如果用十六进制阅读的话，分别为“CAFEBABE”，就会拼出两个单词“CAFEBABE”（咖啡宝贝）。

1994年10月，HotJava和Java平台为公司高层进行演示。1994年，Java1.0a版本已经可以提供下载，但是Java和HotJava浏览器的第一次公开发布却是在1995年5月23日SunWorld大会上进行的。SUN公司的科学指导约翰·盖吉宣告Java技术。这个发布是与网景公司的执行副总裁马克·安德森的惊人发布一起进行的，宣布网景将在其浏览器中包含对Java的支持。1996年1月，Sun公司成立了Java业务集团，专门开发Java技术。

发展简史

- 1995年5月23日，Java语言诞生
- 1996年1月，第一个JDK-JDK1.0诞生
- 1996年4月，10个最主要的操作系统供应商申明将在其产品中嵌入JAVA技术
- 1996年9月，约8.3万个网页应用了JAVA技术来制作
- 1997年2月18日，JDK1.1发布
- 1997年4月2日，JavaOne会议召开，参与者逾一万人，创当时全球同类会议规模之纪录
- 1997年9月，JavaDeveloperConnection社区成员超过十万
- 1998年2月，JDK1.1被下载超过2,000,000次
- 1998年12月8日，JAVA2企业平台J2EE发布
- 1999年6月，SUN公司发布Java的三个版本：标准版（JavaSE,以前是J2SE）、企业版（JavaEE以前是J2EE）和微型版（JavaME，以前是J2ME）
- 2000年5月8日，JDK1.3发布
- 2000年5月29日，JDK1.4发布
- 2001年6月5日，NOKIA宣布，到2003年将出售1亿部支持Java的手机
- 2001年9月24日，J2EE1.3发布
- 2002年2月26日，J2SE1.4发布，自此Java的计算能力有了大幅提升
- 2004年9月30日18:00PM，J2SE1.5发布，成为Java语言发展史上的又一里程碑。为了表示该版本的重要性，J2SE1.5更名为JavaSE5.0
- 2005年6月，JavaOne大会召开，SUN公司公开JavaSE6。此时，Java的各种版本已经更名，以取消其中的数字“2”：J2EE更名为JavaEE，J2SE更名为JavaSE，J2ME更名为JavaME
- 2006年12月，SUN公司发布JRE6.0
- 2009年4月7日GoogleAppEngine开始支持Java
- 2009年04月20日，甲骨文74亿美元收购Sun。取得java的版权。
- 2010年11月，由于甲骨文对于Java社区的不友善，因此Apache扬言将退出JCP。
- 2011年7月28日，甲骨文发布java7.0的正式版。
- 2014年3月19日，甲骨文公司发布java8.0的正式版。

Java 语言与 Java 虚拟机的关系

什么是 Java 虚拟机

Java 虚拟机(Java Virtual Machine,简称 JVM) 是整个 Java 平台的基石，实现硬件与操作系统无关，编译代码后生成出极小体积，保障用户机器免于恶意代码损害。

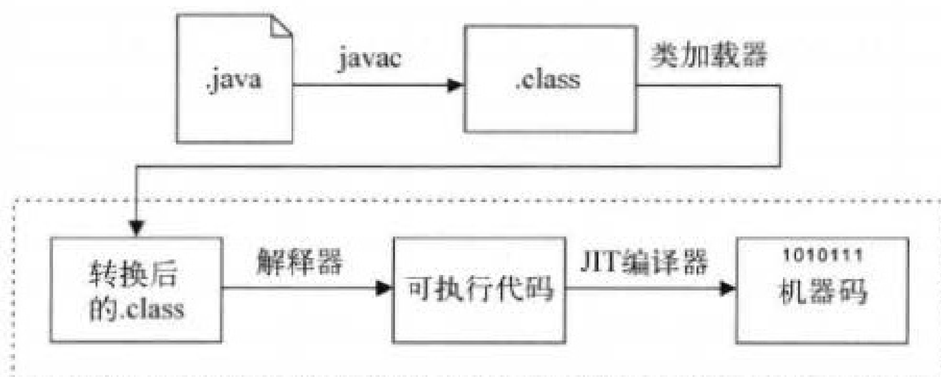
JVM 可以看作是一台抽象的计算机。跟真实的计算机一样，它有自己的指令集以及各种运行时内存区域。使用虚拟机来实现一门程序设计语言有许多合理的理由，业界中流传最为久远的虚拟机可能是 UCSD Pascal 的 P-Code 虚拟机。

第一个 JVM 的原型机是由 Sun 公司实现的，它被用在一种类似 PDA（Personal Digital Assistant，俗称掌上电脑）的手持设备上仿真实现 JVM 指令集。时至今日，Oracle 已有许多 JVM 实现应用于移动设备、桌面电脑、服务器等领域。JVM 并不局限于特定的实现技术、主机硬件和操作系统。它不强求使用解释器来执行程序，也可以通过把自己的指令集编译为实际 CPU 的指令来实现，它可以通过微代码来实现，或者甚至直接实现在 CPU 中。

Java 语言与 JVM 的关系

JVM 与 Java 语言并没有必然的联系，它只与特定的二进制文件格式 class 文件格式所关联。class 文件中包含了 JVM 指令集（或者称为字节码、bytecodes）和符号表，还有一些其他辅助信息。

基于安全方面的考虑，JVM 要求在 class 文件中使用了许多强制性的语法和结构化约束，但任一门功能性语言都可以表示为一个能被 JVM 接收的有效的 class 文件。作为一个通用的、机器无关的执行平台，任何其他语言的实现者都可以将 JVM 作为他们语言的产品交付媒介。



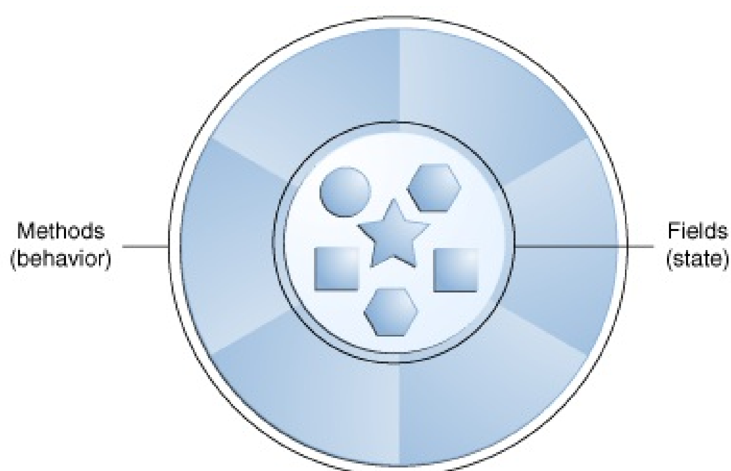
如上图所示，在 Java 编程语言和环境中，即时编译器（JIT compiler，just-in-time compiler）是一个把 Java 的字节码（包括需要被解释的指令的程序）转换成可以直接发送给处理器的指令的程序。当你写好一个 Java 程序后，源语言的语句将由Java 编译器编译成字节码，而不是编译成与某个特定的处理器硬件平台对应的指令代码（比如，Intel 的 Pentium 微处理器或 IBM 的 System/390 处理器）。字节码是可以发送给任何平台并且能在那个平台上运行的独立于平台的代码。

有关 JVNM 的相关内容，可参阅《Java 虚拟机规范》 。

面向对象编程

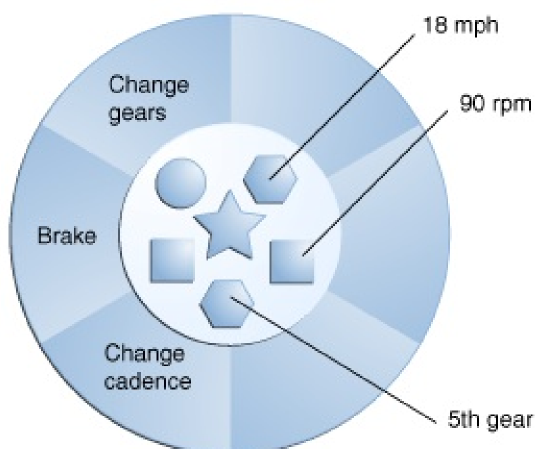
编程的抽象

我们将问题空间中的元素以及它们在方案空间的表示物称作“对象”（Object）。当然，还有一些在问题空间没有对应体的其他对象。通过添加新的对象类型，程序可进行灵活的调整，以便与特定的问题配合。与现实世界的“对象”或者“物体”相比，编程“对 象”与它们也存在共通的地方：它们都有自己的状态(state)和行为(behavior)。比如，狗的状态有名字、颜色等，狗的行为有叫唤、摇尾等。



软件世界中的对象和现实世界中的对象类似，对象存储状态在字段（field）里，而通过方法（methods）暴露其行为。方法对对象的内部状态进行操作，并作为对象与对象之间通信主要机制。隐藏对象内部状态，通过方法进行所有的交互，这个面向对象编程的一个基本原则——数据封装（data encapsulation）。

以单车作为一个对象的建模为例:



通过状态（当前速度，当前踏板节奏，和当前档位），并提供改变这种状态的方法，对象仍然具有如何允许外面的世界使用的控制权。例如，如果自行车只有6个档位，一个方法来改变档位，是可以拒绝任何小于1或比6更大的值。

(1) 所有东西都是对象。可将对象想象成一种新型变量；它保存着数据，但可要求它对自身进行操作。理论上讲，可从要解决的问题身上提出所有概念性的组件，然后在程序中将其表达为一个对象。 (2) 程序是一大堆对象的组合；通过消息传递，各对象知道自己该做些什么。为了向对象发出请求，需向那个对象“发送一条消息”。更具体地讲，可将消息想象为一个调用请求，它调用的是从属于目标对象的一个子例程或函数。 (3) 每个对象都有自己的存储空间，可容纳其他对象。或者说，通过封装现有对象，可制作出新型对象。所以，尽管对象的概念非常简单，但在程序中却可达到任意高的复杂程度。 (4) 每个对象都有一种类型。根据语法，每个对象都是某个“类”的一个“实例”。其中，“类”（Class）是“类型”（Type）的同义词。一个类最重要的特征就是“能将什么消息发给它？”。 (5) 同一类所有对象都能接收相同的消息。由于类型为“圆”（Circle）的一个对象也属于类型为“形状”（Shape）的一个对象，所以一个圆完全能接收形状消息。这意味着可让程序代码统一指挥“形状”，令其自动控制所有符合“形状”描述的对象，其中自然包括 “圆”。这一特性称为对象的“可替换性”，是 OOP 最重要的概念之一。

类（**Class**）的示例

在现实世界中，你经常会发现许多单个对象是同类。有可能是存在其他成千上万的自行车，都是一样的品牌和型号。每个自行车都由相同的一组设计图纸而建成，并因此包含相同的组件。在面向对象的术语，我们说你的自行车被称为自行车对象类（class of objects）的实例（instance）。类就是创建单个对象的设计图纸。

下面是一个 Bicycle (自行车)类的实现：

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

字段 cadence, speed, 和 gear 是对象的状态，方法 changeCadence, changeGear, speedUp 定义了与外界的交互。

你可能已经注意到，Bicycle 类不包含一个 main 方法。这是因为它不是一个完整的应用程序。这是自行车的设计图纸，可能会在应用程序中使用。创建和使用新的 Bicycle对象是应用程序中的其他类的责任。

下面是 BicycleDemo 类，创建两个单独的 Bicycle 对象，并调用其方法：

```
class BicycleDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
         // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();

    }

}
```

执行程序，输出为：

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

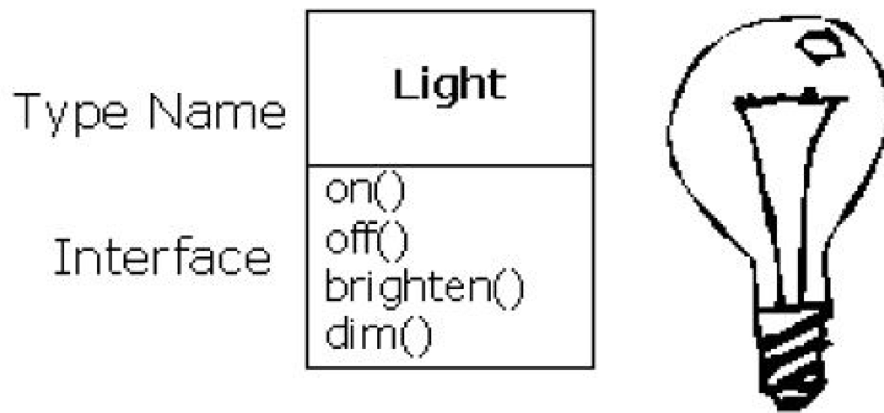
源码

该例子可以在 `com.waylau.essentialjava.object.biycledemo` 包下找到。

对象的接口（Interface）

所有对象——尽管各有特色——都属于某一系列对象的一部分，这些对象具有通用的特征和行为。

每个对象仅能接受特定的请求。我们向对象发出的请求是通过它的“接口”（Interface）定义的，对象的“类型”或“类”则规定了它的接口形式。“类型”与“接口”的等价或对应关系是面向对象程序设计的基础。




```
Light lt = new Light();
lt.on();
```

在这个例子中，类型／类的名称是 Light，Light 对象的名称是 lt ,可向 Light 对象发出的请求包括打开（on）、关闭（off）、变得更明亮（brighten ）或者变得更暗淡（dim）。通过简单地定义一个“引用（reference）”（lt），我们创建了一个 Light 对象，并用 new 关键字来请求那个类新建的对象。为了向对象发送一条消息，我们列出对象名（lt），再用一个句点符号（.）把它同消息名称（on）连接起来。从中可以看出，使用一些预先定义好的类时，我们在程序里采用的代码是非常简单和直观的。

接口的示例

对应自行车的行为，可以定义如下接口：

```
interface Bicycle {

    // wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

实现该接口的类 ACMEBicycle，使用 implements 关键字：

```
class ACMEBicycle implements Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;
```

```
    // The compiler will now require that methods
    // changeCadence, changeGear, speedUp, and applyBrakes
    // all be implemented. Compilation will fail if those
    // methods are missing from this class.

    /*
     * (non-Javadoc)
     *
     * @see
     * com.waylau.essentialjava.interfaceBicycleDemo.Bicycle#changeCadence(int)
     */
    @Override
    public void changeCadence(int newValue) {
        // TODO Auto-generated method stub

    }

    /*
     * (non-Javadoc)
     *
     * @see
     * com.waylau.essentialjava.interfaceBicycleDemo.Bicycle#changeGear(int)
     */
    @Override
    public void changeGear(int newValue) {
        // TODO Auto-generated method stub

    }

    /*
     * (non-Javadoc)
     *
     * @see com.waylau.essentialjava.interfaceBicycleDemo.Bicycle#speedUp(int)
     */
    @Override
    public void speedUp(int increment) {
        // TODO Auto-generated method stub

    }

    /*
     * (non-Javadoc)
     *
     * @see
     * com.waylau.essentialjava.interfaceBicycleDemo.Bicycle#applyBrakes(int)
     */
    @Override
    public void applyBrakes(int decrement) {
        // TODO Auto-generated method stub

    }
```

注： 接口的实现方法前必须添加 public 关键字

源码

该例子可以在 `com.waylau.essentialjava.object.interfacebiycledemo` 包下找到。

包（Package）

包是组织相关的类和接口的命名空间。从概念上讲，类似于计算机上的文件夹，用来将各种文件进行分类。

Java 平台提供了一个巨大的类库（包的集合），该库被称为“应用程序接口”，或简称为“API”。其包代表最常见的与通用编程相关的任务。例如，一个 String 对象包含了字符串的状态和行为; File 对象允许程序员轻松地创建，删除，检查，比较，或者修改文件系统中的文件; Socket 对象允许创建和使用网络套接字;各种 GUI 对象创建图形用户界面。从字面上有数以千计的课程可供选择。作为开发人员只需要专注于特定的应用程序的设计即可，而不是从基础设施建设开始。

对象提供服务

当设计一个程序时，需要将对象想象成一个服务的供应商。对象提供服务给用户，解决不同的问题。

比如，在设计一个图书管理软件，你可能设想一些对象包含了哪些预定义输入，其他对象可能用于图书的统计，一个对象用于打印的校验等。这都需要将一个问题分解成一组对象。

将对象的思考作为服务供应商有一个额外的好处：它有助于改善对象的凝聚力。高内聚（High cohesion） 是软件设计的基本质量：这意味着，一个软件组件的各方面（如对象，尽管这也可以适用于一个方法或一个对象的库）“结合在一起”。在设计对象时经常出现的问题是将太多的功能合并到一个对象里面。例如，在您的支票打印模块，你可以决定你需要知道所有有关格式和打印的对象。你可能会发现，这对于一个对象来说有太多的内容了，那你需要三个或三个以上的对象。一个对象用于查询有关如何打印一张支票的信息目录。一个对象或一组对象可以是知道所有不同类型的打印机的通用打印接口。第三个对象可以使用其他两个对象的服务来完成任务。因此，每个对象都有一套它提供的有凝聚力的服务。良好的面向对象设计，每个对象做好一件事，但不会尝试做太多。

将对象作为服务供应商是一个伟大的简化工具。这不仅在设计过程中是非常有用的，也在当别人试图理解你的代码或重用的对象。如果能看到根据它提供什么样的服务获得对象的值，它可以更容易适应它到设计中。

隐藏实现的细节

为方便后面的讨论，让我们先对这一领域的从业人员作一下分类。从根本上说，大致有两方面的人员涉足面向对象的编程："类创建者"（创建新数据类型的人）以及"客户程序员"（在自己的应用程序中采用现成数据类型的人）。对客户程序员来讲，最主要的目标就是收集一个充斥着各种类的编程"工具箱"，以便快速开发符合自己要求的应用。而对类创建者来说，他们的目标则是从头构建一个类，只向客户程序员开放有必要开放的东西（接口），其他所有细节都隐藏起来。为什么要这样做？隐藏之后，客户程序员就不能接触和改变那些细节，所以原创者不用担心自己的作品会受到非法修改，可确保它们不会对其他人造成影响。

"接口"（Interface）规定了可对一个特定的对象发出哪些请求。然而，必须在某个地方存在着一些代码，以便满足这些请求。这些代码与那些隐藏起来的数据便叫作"隐藏的实现"。一种类型含有与每种可能的请求关联起来的函数。一旦向对象发出一个特定的请求，就会调用那个函数。我们通常将这个过程总结为向对象"发送一条消息"（提出一个请求）。对象的职责就是决定如何对这条消息作出反应（执行相应的代码）。对于任何关系，重要一点是让牵连到的所有成员都遵守相同的规则。创建一个库时，相当于同客户程序员建立了一种关系。对方也是程序员，但他们的目标是组合出一个特定的应用（程序），或者用您的库构建一个更大的库。

若任何人都能使用一个类的所有成员，那么客户程序员可对那个类做任何事情，没有办法强制他们遵守任何约束。即便非常不愿客户程序员直接操作类内包含的一些成员，但倘若未进行访问控制，就没有办法阻止这一情况的发生——所有东西都会暴露无遗。

有两方面的原因促使我们控制对成员的访问。第一个原因是防止程序员接触他们不该接触的东西——通常是内部数据类型的设计思想。若只是为了解决特定的问题，用户只需操作接口即可，毋需明白这些信息。我们向用户提供的实际是一种服务，因为他们很容易就可看出哪些对自己非常重要，以及哪些可忽略不计。

进行访问控制的第二个原因是允许库设计人员修改内部结构，不用担心它会对客户程序员造成什么影响。例如，我们最开始可能设计了一个形式简单的类，以便简化开发。以后又决定进行改写，使其更快地运行。若接口与实现方法早已隔离开，并分别受到保护，就可以很简单的处理。

Java 采用三个显式关键字以及一个隐式关键字来设置类边界：public，private，protected 以及暗示性的 friendly。若未明确指定其他关键字，则默认为后者。friendly 有时也被称为 default。这些关键字的使用和含义都是相当直观的，它们决定了谁能使用后续的定义内容。"public"（公共）意味着后续的定义任何人均可使用。而在另一方面，"private"（私有）意味着除您自己、类型的创建者以及那个类型的内部函数成员，其他任何人都不能访问后续的定义信息。private 在您与客户程序员之间竖起了一堵墙。若有人试图访问私有成员，就会得到一个编译期错误。"friendly "（友好的）涉及"包装"或"封装"（Package）的概念——即 Java 用来构建库的方法。若某样东西是"友好的"，意味着它只能在这个包的范围内使用，所

以这一访问级别有时也叫作“包访问（package access）”。“protected”（受保护的）与“private”相似，只是一个继承的类可访问受保护的成员，但不能访问私有成员。继承的问题不久就要谈到。

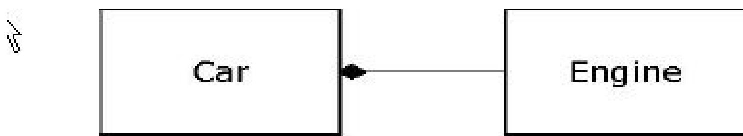
作用域	当前类	同一 **package**	子孙类	其他 **package**
public	√	√	√	√
protected	√	√	√	×
friendly	√	√	×	×
private	√	×	×	×

实现的重用

创建并测试好一个类后，它应（从理想的角度）代表一个有用的代码单位。它要求较多的经验以及洞察力，这样才能使这个类有可能重复使用。

重用是面向对象的程序设计提供的最伟大的一种杠杆。

为重用一个类，最简单的办法是仅直接使用那个类的对象。但同时也能将那个类的一个对象置入一个新类。我们把这叫作“创建一个成员对象”。新类可由任意数量和类型的其他对象构成。这个概念叫作“组合（composition）”（若该组合是动态发生，则也成为“聚合（aggregation）”）——在现有类的基础上组合为一个新类。有时，我们也将组合称作“包含（has-a）”关系，比如“一辆车包含了一个变速箱”。



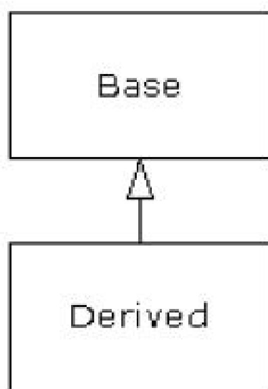
对象的组合具有极大的灵活性。新类的“成员对象”通常设为“私有”（Private），使用这个类的客户程序员不能访问它们。这样一来，我们可在不干扰客户代码的前提下，从容地修改那些成员。也可以在“运行期”更改成员，这进一步增大了灵活性。后面要讲到的“继承”并不具备这种灵活性，因为编译器必须对通过继承创建的类加以限制。

继承虽然重要，但作为新加入这新建类的时候，首先应考虑“组合”对象；这样做显得更加简单和灵活。利用对象的组合，我们的设计可保持清爽。

继承

我们费尽心思做出一种数据类型后，假如不得不又新建一种类型，令其实现大致相同的功能，那会是一件非常令人灰心的事情。但若能利用现成的数据类型，对其进行“克隆”，再根据情况进行添加和修改，情况就显得理想多了。“继承”正是针对这个目标而设计的。但继承并不完全等价于克隆。在继承过程中，若原始类（正式名称叫作基础类、超类或父类）发生了变化，修改过的“克隆”类（正式名称叫作派生类或者继承类或者子类）也会反映出这种变化。在 Java 语言中，继承是通过 extends 关键字实现的 使用继承时，相当于创建了一个新类。这个新类不仅包含了现有类型的所有成员（尽管private 成员被隐藏起来，且不能访问），但更重要的是，它复制了基础类的接口。也就是说，可向基础类的对象发送的所有消息亦可原样发给衍生类的对象。根据可以发送的消息，我们能知道类的类型。这意味着衍生类具有与基础类相同的类型！

由于基础类和派生类具有相同的接口，所以那个接口必须进行特殊的设计。也就是说，对象接收到一条特定的消息后，必须有一个“方法”能够执行。若只是简单地继承一个类，并不做其他任何事情，来自基础类接口的方法就会直接照搬到派生类。这意味着派生类的对象不仅有相同的类型，也有同样的行为，这一后果通常是我们不愿见到的。



有两种做法可将新得的派生类与原来的基础类区分开。第一种做法十分简单：为派生类添加新函数（功能）。这些新函数并非基础类接口的一部分。进行这种处理时，一般都是意识到基础类不能满足我们的要求，所以需要添加更多的函数。这是一种最简单、最基本的继承用法，大多数时候都可完美地解决我们的问题。然而，事先还是要仔细调查自己的基础类是否真的需要这些额外的函数。

尽管 extends 关键字暗示着我们要为接口“扩展”新功能，但实情并非肯定如此。为区分我们的新类，第二个办法是改变基础类一个现有函数的行为。我们将其称作“改善”那个函数。为改善一个函数，只需为衍生类的函数建立一个新定义即可。我们的目标是：“尽管使用的函数接口未变，但它的新版本具有不同的表现”。

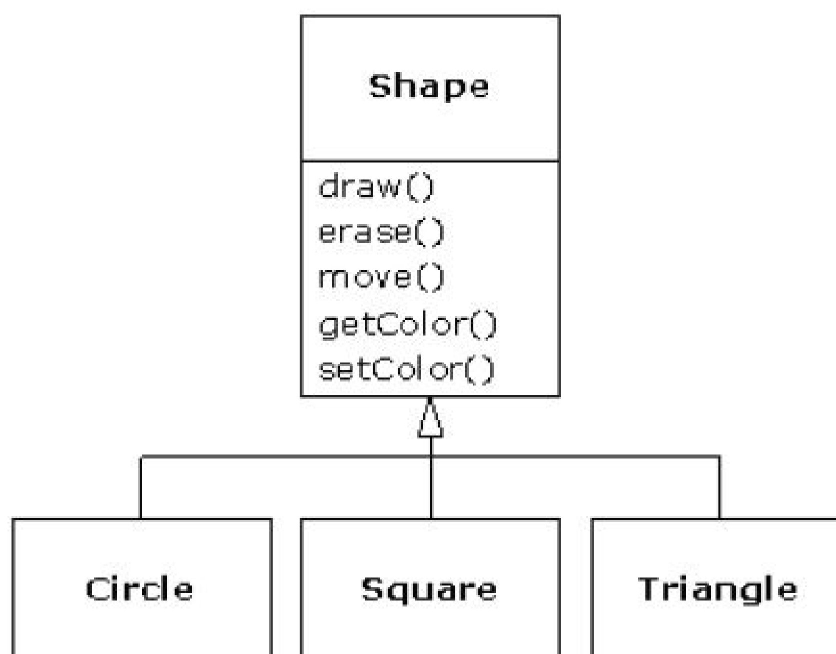
针对继承可能会产生这样的一个争论：继承只能改善原基础类的函数吗？若答案是肯定的，则派生类型就是与基础类完全相同的类型，因为都拥有完全相同的接口。这样造成的结果就是：我们完全能够将派生类的一个对象换成基础类的一个对象！可将其想象成一种“纯替换”。

在某种意义上，这是进行继承的一种理想方式。此时，我们通常认为基础类和派生类之间存在一种“等价”关系——因为我们可以理直气壮地说：“圆就是一种几何形状”。为了对继承进行测试，一个办法就是看看自己是否能把它们套入这种“等价”关系中，看看是否有意义。

但在许多时候，我们必须为派生类型加入新的接口元素。所以不仅扩展了接口，也创建了一种新类型。这种新类型仍可替换成基础类型，但这种替换并不是完美的，因为不可在基础类里访问新函数。我们将其称作“类似”关系；新类型拥有旧类型的接口，但也包含了其他函数，所以不能说它们是完全等价的。举个例子来说，让我们考虑一下制冷机的情况。假定我们的房间连好了用于制冷的各种控制器；也就是说，我们已拥有必要的“接口”来控制制冷。现在假设机器出了故障，我们把它换成一台新型的冷、热两用空调，冬天和夏天均可使用。冷、热空调“类似”制冷机，但能做更多的事情。由于我们的房间只安装了控制制冷的设备，所以它们只限于同新机器的制冷部分打交道。新机器的接口已得到了扩展，但现有的系统并不知道除原始接口以外的任何东西。

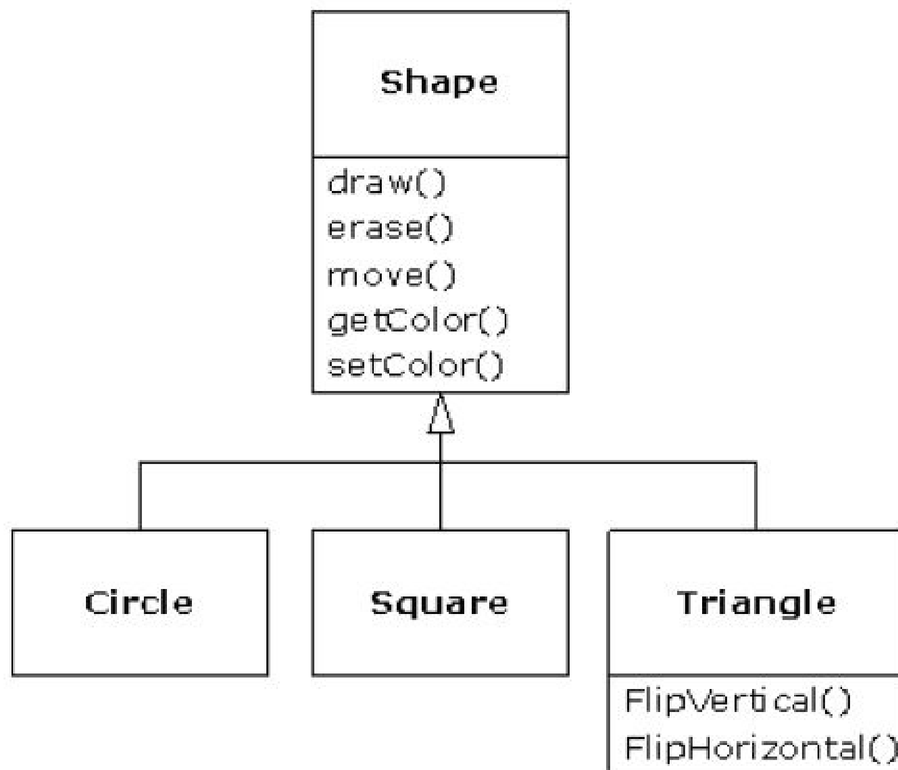
认识了等价与类似的区别后，再进行替换时就会有把握得多。尽管大多数时候“纯替换”已经足够，但您会发现在某些情况下，仍然有明显的理由需要在衍生类的基础上增添新功能。通过前面对这两种情况的讨论，相信大家已心中有数该如何做。

另外一个是 “shape” 示例，基本类型是 “shape”，每个 shape 都有尺寸、颜色、位置等。每个 shape 都可以画、清除、移动、上色等。特定 shape 的派生类型 circle, square, triangle 等，都可能有自己的特征和行为。例如，某些 shape 可以翻转，有些行为可能会有所不同，比如，当你要计算一个 shape 的面积。

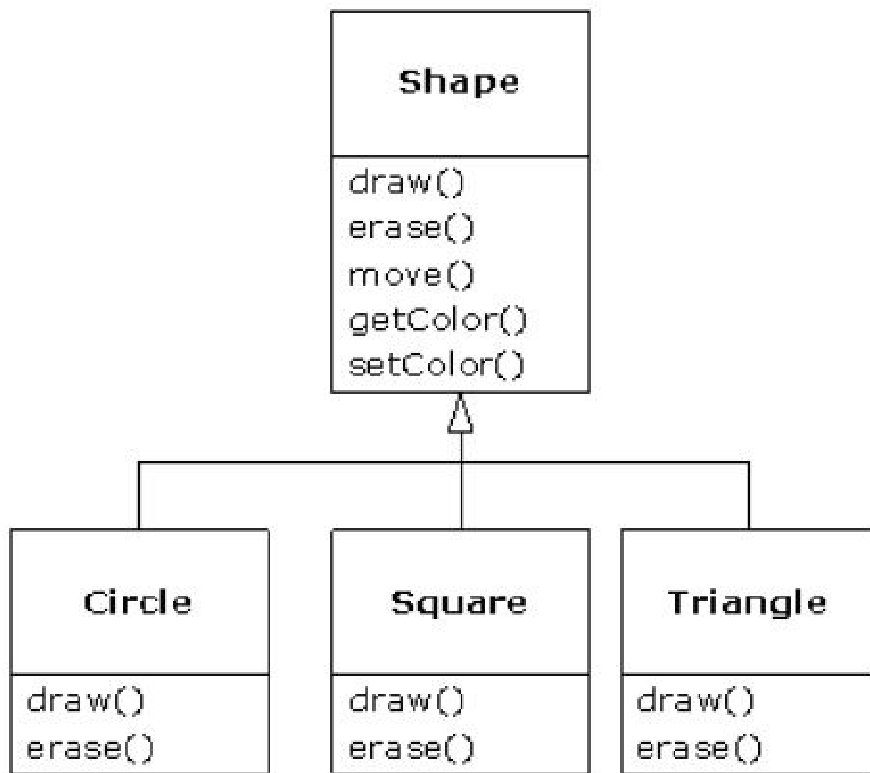


你有2种方法来区分原来的基类和新派生类。第一个非常简单：你只需向派生类添加新的方法。这些新方法不是基类接口的一部分。这意味着基本类没有你所希望的方法，所以你增加了更多的方法。这个是简单而原始的继承使用，有时，你的问题的完美解决方案。然而，另

一种可能性，你的基础类可能也需要这些额外的方法。在面向对象程序设计中，经常会出现这种发现和迭代。



虽然继承可能有时意味着（特别是 Java，继承的关键字就是“extends（扩展）”）你将添加新的方法到接口，这不一定总是对的。第二种更重要的方式来区分你的新类是改变现有基类方法的行为。这被称为方法“覆盖（overriding）”。

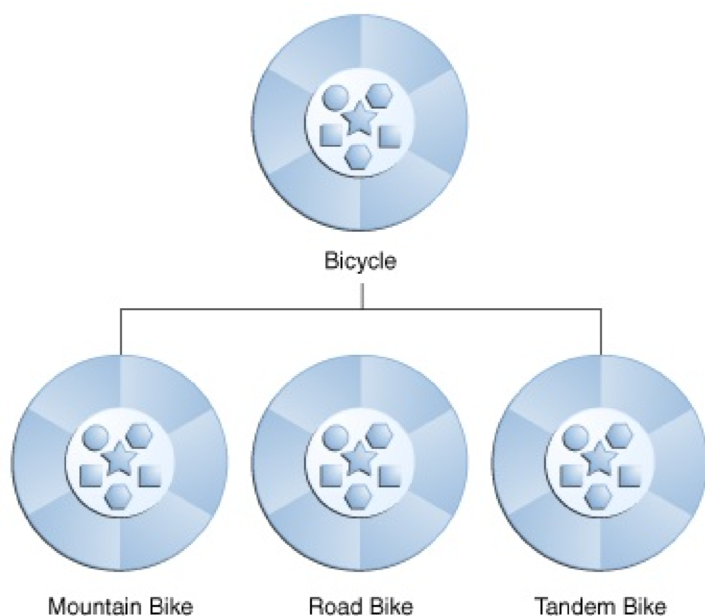


覆盖的方法，您只需为派生类中的方法创建一个新的定义。“使用的是相同的接口方法，但在新类型里做不同的事情”。

继承的示例

不同种类的对象往往有一定量的在共同点。例如，山地自行车(Mountain Bike)，公路自行车(Road Bike)和双人自行车(Tandem Bike)，所有的自行车都有共同的特点：当前目前的速度，当前踏板节奏，当前档位。然而，每一个还定义了额外的功能，使他们不同：双人自行车有两个座位和两套车把;公路自行车有下降车把;一些山地自行车有一个附加的链环，使得他们具有一个较低的齿轮比。

面向对象的编程允许类从其他类继承常用的状态和行为。在这个例子中，自行车(Bicycle)现在变成山地车，公路自行车和双人自行车的超类。在 Java 编程语言中，每一个类被允许具有一个直接超类，每个超具有无限数量的子类的潜力：



继承使用 extends 关键字：

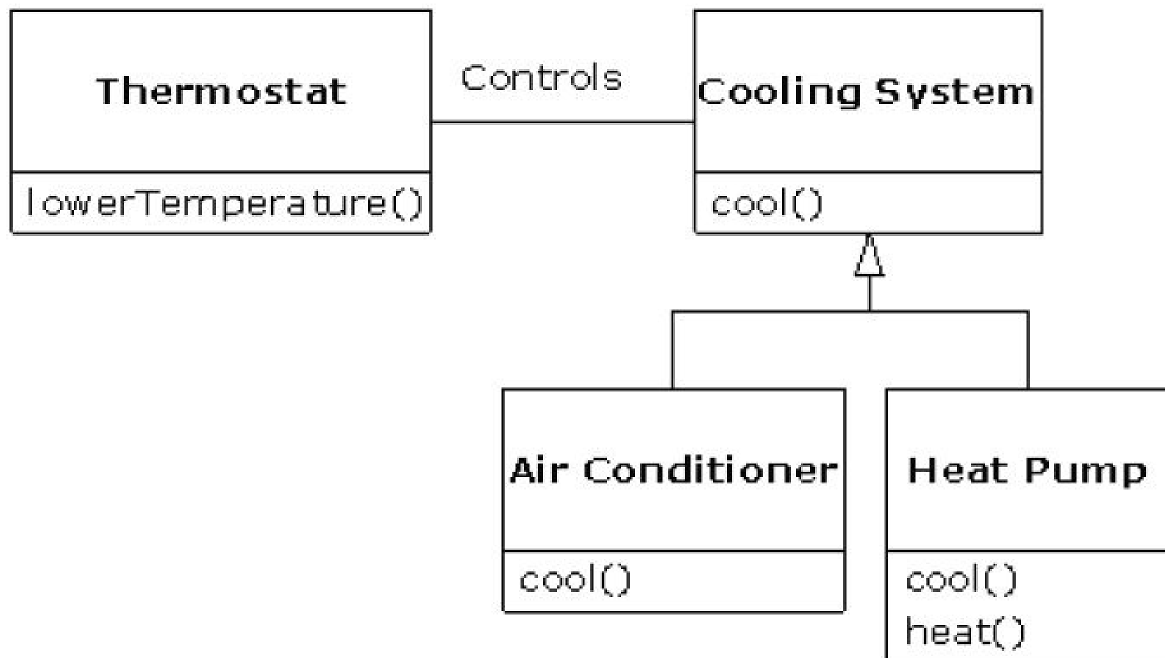
```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here
}
```

is-a 和 is-like-a 的关系

有个讨论是关于继承的：继承只应该重写基类方法（不添加基类中没有的新的方法）？这将意味着派生类完全是同一类的基类，因为它有完全相同的接口。作为一个结果，您可以完全用基类的对象替换派生类的对象。这可以被认为是纯粹的替代，它通常被称为替代原则。从这个意义上说，这是对待继承的理想方法。这个就是 is-a 关系，可以说，“圆是一种形状（A circle is a shape）”。这是对于测试确定一些类是否是 is-a 关系是非常有用的。

有时，必须将新的接口元素添加到派生类型，从而扩展接口。新的类型仍然可以被替换为基类型，但替换并不是完美的，因为你的新方法是不可从基类型访问的。这可以被描述为一个 islike-a 关系。新类型拥有旧类型的接口，但它也包含其他的方法，所以你不能真的说它是完全相同的。例如，考虑一个空调。假设你的房子与所有的冷却控制连接，也就是说，它有一个接口，允许你控制冷却。想象一下，空调坏了，你用一个热泵替换它，它可以加热和冷却。热泵像一个空调，但它可以做更多。因为你的房子的控制系统的设计只是为了控制冷却，它被限制在只能与新的对象的冷却部分通信。新对象的接口已扩展，但现有的系统不知道除了原始接口以外的任何事情。



当然，一旦你看到这个设计，就很清楚，基本的“冷却系统(cooling system)”是不够的，应该重新命名为“温度控制系统(temperature control system)”，这样也可以包括加热。然而，这个图是一个可以在现实世界中发生的例子。

替代原则这种方法（纯替代）不是唯一的方式，有时你必须向派生类的接口添加新的方法，使得设计更加合理。

多态性（Polymorphism）

什么是多态

面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

多态存在的三个必要条件：

- 要有继承
- 要有重写
- 父类引用指向子类对象。

多态的好处：

1. 可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。
2. 可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。
3. 接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。图中超类 Shape 规定了两个实现多态的接口方法，computeArea() 以及 computeVolume()。子类，如 Circle 和 Sphere 为了实现多态，完善或者覆盖这两个接口方法。
4. 灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。
5. 简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

源码

本章例子的源码，可以在 `com.waylau.essentialjava.object` 包下找到。

语言基础

本章介绍 Java 的语言基础。

变量（Variable）

前文也介绍了，对象的状态是存储在字段里面

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Java 里面的变量包含如下类型：

- 实例变量/非静态字段(Instance Variables/Non-Static Fields):从技术上讲，对象存储他们的个人状态在“非静态字段”，也就是没有 static 关键字声明的字段。非静态字段也被称为实例变量，因为它们的值对于类的每个实例来说是唯一的（换句话说，就是每个对象）；自行车的当前速度独立于另一辆自行车的当前速度。
- 类变量/静态字段(Class Variables/Static Fields) ：类变量是用 static 修饰符声明的字段，也就是告诉编译器无论类被实例化多少次，这个变量的存在，只有一个副本。特定种类自行车的齿轮数目的字段可以被标记为 static，因为相同的齿轮数量将适用于所有情况。代码 `static int numGears = 6;` 将创建一个这样的静态字段。此外，关键字 final 也可以加入，以指示的齿轮的数量不会改变。
- 局部变量(Local Variables)：类似于对象存储状态在字段里，方法通常会存放临时状态在局部变量里。语法与局部变量的声明类似（例如，`int count = 0;`）。没有特殊的关键字来指定一个变量是否是局部变量，是由该变量声明的位置决定的。局部变量是类的方法中的变量。
- 参数（Parameters）：前文的例子中经常可以看到 `public static void main(String[] args)` ，这里的 args 变量就是这个方法参数。要记住的重要一点是，参数都归类为“变量（variable）”而不是“字段（field）”。

如果我们谈论的是“一般的字段”（不包括局部变量和参数），我们可以简单地说“字段”。如果讨论适用于上述所有情况，我们可以简单地说“变量”。如果上下文要求一个区别，我们将使用特定的术语（静态字段，局部变量，等等）。你也偶尔会看到使用术语“成员（member）”。类型的字段、方法和嵌套类型统称为它的成员。

命名

每一个编程语言都有它自己的一套规则和惯例的各种名目的，Java 编程语言对于命名变量的规则和惯例可以概括如下：

- 变量名称是区分大小写的。变量名可以是任何合法的标识符 - 无限长度的 Unicode字母和数字，以字母，美元符号 `$` ，或下划线 `_` 开头。但是惯例上推荐使用字母开头，而不

是 `$` 或 `_` 。此外，按照惯例，美元符号从未使用过的。您可能会发现一些情况，自动生成的名称将包含美元符号，但你的变量名应该始终避免使用它。类似的约定存在下划线，不鼓励用“_”作为变量名开头。空格是不允许的。

- 随后的字符可以是字母，数字，美元符号，或下划线字符。惯例同样适用于这一规则。为变量命名，尽量是完整的单词，而不是神秘的缩写。这样做会使你的代码更容易阅读和理解，比如 cadence、speed 和 gear 会比缩写 c、s 和 g 更直观。同时请记住，您选择的名称不能是关键字或保留字。
- 如果您选择的名称只包含一个词，拼写单词全部小写字母。如果它由一个以上的单词，每个后续单词的第一个字母大写，如 gearRatio 和 currentGear。如果你的变量存储一个恒定值，使用 `static final int NUM_GEARS = 6` ，每个字母大写，并用以下划线分隔后续字符。按照惯例，下划线字符永远不会在其他地方使用。

详细的命名规范，可以参考《Java 编码规范》 。

基本数据类型（Primitive Data Types）

Java 是静态类型（statically-typed）的语言，必须先声明再使用。基本数据类型之间不会共享状态。

主要有8种基本数据类型：

primitive	numeric	byte
		short
		int
		long
		char
	floating-point	float
		double
		looean
reference		class type
		interface type
		array type
		null type

byte

byte 由1个字节8位表示，是最小的整数类型。主要用于节省内存空间关键。当操作来自网络、文件或者其他 IO 的数据流时，byte类型特别有用。取值范围为:[-128, 127]. byte 的默认值为 (byte)0,如果我们试图将取值范围外的值赋给 byte类型变量，则会出现编译错误，例如 `byte b = 128;` 这个语句是无法通过编译的。一个有趣的问题，如果我们有个方法： public void test(byte b)。试图这么调用这个方法是错误的: test(0); 编译器会报错，类型不兼容！！！我们记得 `byte b =0;` 这是完全没有问题的，为什么在这里就出错啦？

这里涉及到一个叫字面值（literal）的问题，字面值就是表面上的值，例如整型字面值在源代码中就是诸如 5 ， 0， -200这样的。如果整型子面子后面加上L或者l，则这个字面值就是long 类型，比如：1000L代表一个 long 类型的值。如果不加L或者l，则为 int 类型。基本类型当中的byte short int long都可以通过不加L的整型字面值（我们就称作int字面值吧）来创建，例如 byte b = 100； short s = 5；对于long类型，如果大小超出int所能表示的范围（32 bits），则必须使用L结尾来表示。整型字面值可以有不同的表示方式：16进制【0X or 0x】、10进制【nothing】、八进制【0】、2进制【0B or 0b】等，二进制字面值是JDK 7以后才有的功能。在赋值操作中，int字面值可以赋给byte short int long，Java语言会自动处理好这个过程。如果方法调用时不一样，调用test（0）的时候，它能匹配的方法是test（int），当然不能匹配test（byte）方法，至于为什么Java没有像支持赋值操作那样支持方法调用，不得而知。注意区别包装器与原始类型的自动转换（anto-boxing，auto-unboxing）。 `byte d = 'A'；` 也是合法的，字符字面值可以自动转换成16位的整数。 对byte类型进行数学运算时，会自动提升为int类型，如果表达式中有double或者float等类型，也是自动提升。所以下面的代码是错误的：

```
byte t s1 = 100;
byte s2 = 'a';
byte sum = s1 + s2;//should cast by (byte)
```

short

用 16 位表示，取值范围为：[- 2^15, 2^15 - 1]。short可能是最不常用的类型了。可以通过整型字面值或者字符字面值赋值，前提是不超出范围（16 bit）。short类型参与运算的时候，一样被提升为int或者更高的类型。（顺序为 byte short int long float double).

int

32 bits, [- 2^31, 2^31 - 1].有符号的二进制补码表示的整数。常用语控制循环，注意byte 和 short在运算中会被提升为int类型或更高。Java 8以后，可以使用int类型表示无符号32位整数[0, 2^31 - 1]。

long

64 bits， [- 2^63, 2^63 - 1,默认值为0L].当需要计算非常大的数时，如果int不足以容纳大小，可以使用long类型。如果long也不够，可以使用BigInteger类。

char

16 bits, [0, 65535], [0, 2^16 -1],从'\u0000'到'\uffff'。无符号，默认值为'\u0000'。Java使用Unicode字符集表示字符，Unicode是完全国际化的字符集，可以表示全部人类语言中的字符。Unicode需要16位宽，所以Java中的char类型也使用16 bit表示。 赋值可能是这样的：

```
char ch1 = 88;
char ch2 = 'A';
```

ASCII字符集占用了Unicode的前127个值。之所以把char归入整型，是因为Java为char提供算术运算支持，例如可以ch2++;之后ch2就变成Y。当char进行加减乘除运算的时候，也被转换成int类型，必须显式转化回来。

float

使用32 bit表示，对应单精度浮点数，遵循IEEE 754规范。运行速度相比double更快，占内存更小，但是当数值非常大或者非常小的时候会变得不精确。精度要求不高的时候可以使用float类型，声明赋值示例：

```
float f1 =10;
f1 = 10L;
f1 = 10.0f;  //f1 = 10.0;默认为double
```

可以将byte、short、int、long、char赋给float类型，java自动完成转换。

double

64为表示，将浮点子面子赋给某个变量时，如果不显示在字面值后面加f或者F，则默认为double类型。java.lang.Math中的函数都采用double类型。

如果double和float都无法达到想要的精度，可以使用BigDecimal类。

boolean

boolean类型只有两个值true和false，默认为false。boolean与是否为0没有任何关系，但是可以根据想要的逻辑进行转换。许多地方都需要用到boolean类型。

除了上面列出的八种原始数据类型，Java 编程语言还提供了 java.lang.String用于字符串的特殊支持。双引号包围的字符串会自动创建一个新的 String 对象，例如 `String s = "this is a string";` 。String 对象是不可变的（immutable），这意味着一旦创建，它们的值不能改变。String 类不是技术上的原始数据类型，但考虑由语言所赋予的特殊支持，你可能会倾向于认为它是这样的。更多关于 String 类的细节，可以参阅简单数据对象（Simple Data Objects）。

默认值

在字段声明时，有时并不必要分配一个值。字段被声明但尚未初始化时，将会由编译器设置一个合理的默认值。一般而言，根据数据类型此的不同，默认将为零或为 null。但良好的的编程风格不应该依赖于这样默认值。

下面的图表总结了上述数据类型的默认值。

数据类型	字段默认值
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (或任何对象)	null
boolean	false

局部变量（Local Variable）略有不同，编译器不会指定一个默认值未初始化的局部变量。如果你不能初始化你声明的局部变量，那么请确保使用之前，给它分配一个值。访问一个未初始化的局部变量会导致编译时错误。

字面值（Literal）

在 Java 源代码中，字面值用于表示固定的值（fixed value），直接展示在代码里，而不需要计算。数值型的字面值是最常见的，字符串字面值可以算是一种，当然也可以把特殊的 null 当做字面值。字面值大体上可以分为整型字面值、浮点字面值、字符和字符串字面值、特殊字面值。

整型字面值

从形式上看是整数的字面值归类为整型字面值。例如： 10, 100000L, 'B'、0XFF这些都可以称为字面值。整型字面值可以用十进制、16、8、2进制来表示。十进制很简单，2、8、16进制的表示分别在最前面加上0B（0b）、0、0X（0x）即可，当然基数不能超出进制的范围，比如09是不合法的，八进制的基数只能到7。一般情况下，字面值创建的是int类型，但是int字面值可以赋值给byte short char long int，只要字面值在目标范围以内，Java会自动完成转换，如果试图将超出范围的字面值赋给某一类型（比如把128赋给byte类型），编译通不过。而如果想创建一个int类型无法表示的long类型，则需要在字面值最后面加上L或者l。通常建议使用容易区分的L。所以整型字面值包括int字面值和long字面值两种。

- 十进制(Decimal)：其位数由数字0~9组成;这是您每天使用的数字系统
- 十六进制(Hexadecimal)：其位数由数字0到9和字母A至F的组成
- 二进制(Binary)：其位数由数字0和1的（可以在 Java SE 7 和更高版本创建二进制字面值）

下面是使用的语法：

```
// The number 26, in decimal
int decVal = 26;
//  The number 26, in hexadecimal
int hexVal = 0x1a;
// The number 26, in binary
int binVal = 0b11010;
```

浮点字面值

浮点字面值简单的理解可以理解为小数。分为float字面值和double字面值，如果在小数后面加上F或者f，则表示这是个float字面值，如11.8F。如果小数后面不加F（f），如10.4。或者小数后面加上D（d），则表示这是个double字面值。另外，浮点字面值支持科学技术法(E 或 e)表示。下面是一些例子：

```
double d1 = 123.4;
// same value as d1, but in scientific notation
double d2 = 1.234e2;
float f1  = 123.4f;
```

字符及字符串字面值

Java 中字符字面值用单引号括起来，如 `@` ， `1` 。所有的UTF-16字符集都包含在字符字面值中。不能直接输入的字符，可以使用转义字符，如 `\n` 为换行字符。也可以使用八进制或者十六进制表示字符，八进制使用反斜杠加3位数字表示，例如 `\141` 表示字母a。十六进制使用 `\u` 加上4为十六进制的数表示，如 `\u0061` 表示字符a。也就是说，通过使用转义字符，可以表示键盘上的有的或者没有的所有字符。常见的转义字符序列有：

`\ddd(八进制)` 、 `\uxxxx(十六进制Unicode字符)` 、 `\'（单引号）` 、 `\"（双引号）` 、 `\\ （反斜杠）`
`\r（回车符） \n（换行符） \f（换页符） \t（制表符） \b（回格符）`

字符串字面值则使用双引号，字符串字面值中同样可以包含字符字面值中的转义字符序列。字符串必须位于同一行或者使用+运算符，因为 Java 没有续行转义序列。

在数值型字面值中使用下划线

Java SE 7 开始，可以在数值型字面值中使用下划线。但是下划线只能用于分隔数字，不能分隔字符与字符，也不能分隔字符与数字。例如 `int x = 123_456_789` ，在编译的时候，下划线会自动去掉。可以连续使用下划线，比如 `float f = 1.22___33__44` 。二进制或者十六进制的字面值也可以使用下划线，记住一点，下划线只能用于数字与数字之间，初次以外都是非法的。例如1._23是非法的，_123、11000_L都是非法的。

正确的用法：

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi =  3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

非法的用法：

```
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi1 = 3_.1415F;
// Invalid: cannot put underscores
// adjacent to a decimal point
float pi2 = 3._1415F;
// Invalid: cannot put underscores
// prior to an L suffix
long socialSecurityNumber1 = 999_99_9999_L;

// Invalid: cannot put underscores
// At the end of a literal
int x2 = 52_;

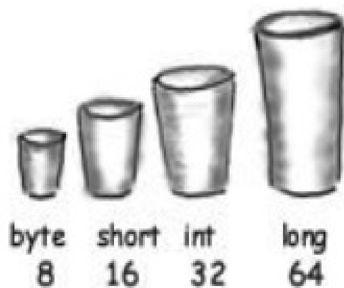
// Invalid: cannot put underscores
// in the 0x radix prefix
int x4 = 0_x52;
// Invalid: cannot put underscores
// at the beginning of a number
int x5 = 0x_52;
// Invalid: cannot put underscores
// at the end of a number
int x7 = 0x52_;
```

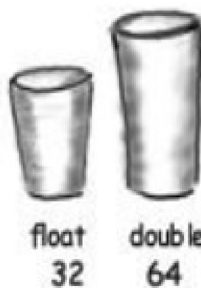
基本类型之间的转换

我们看到，将一种类型的值赋给另一种类型是很常见的。在 Java 中，boolean 类型与所有其他7种类型都不能进行转换，这一点很明确。对于其他7中数值类型，它们之间都可以进行转换，但是可能会存在精度损失或者其他一些变化。转换分为自动转换和强制转换。对于自动转换（隐式），无需任何操作，而强制类型转换需要显式转换，即使用转换操作符（type）。首先将7种类型按下面顺序排列一下：

```
byte <（short=char）< int < long < float < double
```

如果从小转换到大，可以自动完成，而从大到小，必须强制转换。short 和 char 两种相同类型也必须强制转换。







自动转换

自动转换时发生扩宽（widening conversion）。因为较大的类型（如int）要保存较小的类型（如byte），内存总是足够的，不需要强制转换。如果将字面值保存到 byte、short、char、long 的时候，也会自动进行类型转换。注意区别，此时从 int（没有带L的整型字面值为int）到 byte/short/char 也是自动完成的，虽然它们都比int 小。在自动类型转化中，除了以下几种情况可能会导致精度损失以外，其他的转换都不能出现精度损失。

- int--> float
- long--> float
- long--> double
- float -->double without strictfp

除了可能的精度损失外，自动转换不会出现任何运行时（run-time）异常。

强制类型转换

如果要把大的转成小的，或者在 short 与 char 之间进行转换，就必须强制转换，也被称作缩小转换（narrowing conversion）,因为必须显式地使数值更小以适应目标类型。强制转换采用转换操作符（）。严格地说，将 byte 转为 char 不属于 （narrowing conversion），因为从

byte 到 char 的过程其实是 byte-->int-->char，所以 widening 和 narrowing 都有。强制转换除了可能的精度损失外，还可能使模（overall magnitude）发生变化。强制转换格式如下：（target-type） value

```
int a=257;
byte b;
b = (byte)a;//1
```

如果整数的值超出了 byte 所能表示的范围，结果将对 byte 类型的范围取余数。例如a=256超出了byte的[-128,127]的范围，所以将257除以byte的范围（256）取余数得到b=1；需要注意的是，当a=200时，此时除了256取余数应该为-56，而不是200. 将浮点类型赋给整数类型的时候，会发生截尾（truncation）。也就是把小数的部分去掉，只留下整数部分。此时如果整数超出目标类型范围，一样将对目标类型的范围取余数。

7中基本类型转换总结如下图：

To \ From	byte	short	char	int	long	float	double
byte	-	(byte)	(byte)	(byte)	(byte)	(byte)	(byte)
short		-	(short)	(short)	(short)	(short)	(short)
char		(char)	-	(char)	(char)	(char)	(char)
int				-	(int)	(int)	(int)
long					-	(long)	(long)
float						-	(float)
double							-

赋值及表达式中的类型转换：

字面值赋值

在使用字面值对整数赋值的过程中，可以将int literal赋值给byte short char int，只要不超出范围。这个过程中的类型转换时自动完成的，但是如果你试图将long literal赋给byte，即使没有超出范围，也必须进行强制类型转换。例如 byte b = 10L；是错的，要进行强制转换。

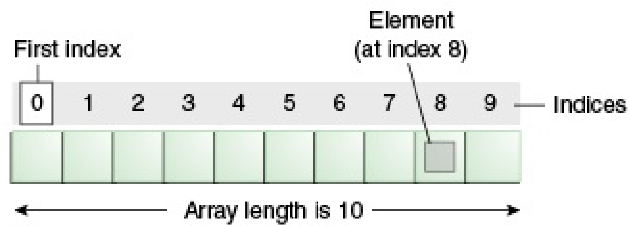
表达式中的自动类型提升

除了赋值以外，表达式计算过程中也可能发生一些类型转换。在表达式中，类型提升规则如下：

- 所有byte/short/char都被提升为int。
- 如果有一个操作数为long，整个表达式提升为long。float和double情况也一样。

数组（Array）

数组是一个容器对象，保存一个固定数量的单一类型的值。当数组创建时，数组的长度就确定了。创建后，其长度是固定的。下面是一个例子：



数据里面的每个项称为元素（element），每个元素都用一个数组下标（index）关联，下标是从 0 开始，如上图所示，第9个元素的下标是 8：

ArrayDemo 的示例：

```
class ArrayDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        // declares an array of integers
        int[] anArray;

        // allocates memory for 10 integers
        anArray = new int[10];

        // initialize first element
        anArray[0] = 100;
        // initialize second element
        anArray[1] = 200;
        // and so forth
        anArray[2] = 300;
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

输出为：

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

声明引用数组的变量

声明数组的类型，如下：

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

也可以将中括号放数组名称后面（但不推荐）

```
// this form is discouraged
float anArrayOfFloats[];
```

创建、初始化和访问数组

ArrayDemo 的示例说明了创建、初始化和访问数组的过程。可以用下面的方式，简化 创建、初始化数组

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

数组里面可以声明数组，即，多维数组 （multidimensional array）。如 MultiDimArrayDemo 例子：

```
class MultiDimArrayDemo {
    /**
     * @param args
     */
    public static void main(String[] args) {
        String[][] names = { { "Mr. ", "Mrs. ", "Ms. " }, { "Smith", "Jones" } };
        // Mr. Smith
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Jones
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

输出为：

```
Mr. Smith
Ms. Jones
```

最后，可以通过内建的 length 属性来确认数组的大小

```
System.out.println(anArray.length);
```

复制数组

System 类有一个 arraycopy 方法用于数组的有效复制：

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

下面是一个例子 ArrayCopyDemo，

```
class ArrayCopyDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e'
, 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

程序输出为：

```
caffein
```

数组操作

Java SE 提供了一些数组有用的操作。 ArrayCopyOfDemo 例子：

```
class ArrayCopyOfDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e'
, 'd' };

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}
```

可以看到，使用 java.util.Arrays.copyOfRange 方法，代码量减少了。

其他常用操作还包括：

- binarySearch ： 用于搜索
- equals ： 比较两个数组是否相等
- fill : 填充数组
- sort : 数组排序，在 Java SE 8 以后，可以使用 parallelSort 方法，在多处理器系统的大数组并行排序比连续数组排序更快。

源码

该例子可以在 `com.waylau.essentialjava.array.arraydemo` 包下找到。

运算符

靠近表顶部的运算符，其优先级最高。具有较高优先级的运算符在相对较低的优先级的运算符之前被评估。在同一行上的运算符具有相同的优先级。当在相同的表达式中出现相同优先级的运算符时，必须首先对该规则进行评估。除了赋值运算符外，所有二进制运算符进行评估从左到右，赋值操作符是从右到左。

运算符优先级表：

运算符	优先级		
后缀（postfix）	`expr++ expr--`		
一元运算（unary）	`++expr --expr +expr -expr ~ !`		
乘法（multiplicative）	`* / %`		
加法（additive）	`+ -`		
移位运算（shift）	`<< >> >>>`		
关系（relational）	`< > <= >= instanceof`		
相等（equality）	`== !=`		
与运算（bitwise AND）	`&`		
异或运算（bitwise exclusive OR）	`^`		
或运算（bitwise inclusive OR）	`	`	
逻辑与运算（logical AND）	`&&`		
逻辑或运算（logical OR）	`		`
三元运算（ternary）	`? :`		
赋值运算（assignment）	`= += -= *= /= %= &= ^=	= <<= >>= >>>=`	

赋值运算符

赋值运算符是最常用和最简单的运算符就是 `=`，如下：

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

该运算符也用于对象的引用关联。

算术运算符

算术运算符包括：

运算符 | 描述 |:----:| ----| |+ | 加 (也用于 String 的连接)| |- | 减| |* | 乘| |/ | 除| |% | 取余|

ArithmeticDemo 的例子

```
class ArithmeticDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int result = 1 + 2;
        // result is now 3
        System.out.println("1 + 2 = " + result);
        int original_result = result;

        result = result - 1;
        // result is now 2
        System.out.println(original_result + " - 1 = " + result);
        original_result = result;

        result = result * 2;
        // result is now 4
        System.out.println(original_result + " * 2 = " + result);
        original_result = result;

        result = result / 2;
        // result is now 2
        System.out.println(original_result + " / 2 = " + result);
        original_result = result;

        result = result + 8;
        // result is now 10
        System.out.println(original_result + " + 8 = " + result);
        original_result = result;

        result = result % 7;
        // result is now 3
        System.out.println(original_result + " % 7 = " + result);
    }

}
```

输出为：

```
1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3
```

`+` 用于字符串连接的例子 ConcatDemo ：

```
class ConcatDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String firstString = "This is";
        String secondString = " a concatenated string.";
        String thirdString = firstString+secondString;
        System.out.println(thirdString);
    }
}
```

一元操作

一元运算符只需要一个操作数。

运算符 | 描述 |:----:| ----| |+ | 加运算;指正值| |- | 减运算符;表达成负值| |++ | 递增运算符;递增值1| |-- | 递减运算符;递减值1| | ! | 逻辑补运算;反转一个布尔值|

下面是 UnaryDemo 的示例：

```
class UnaryDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int result = +1;
        // result is now 1
        System.out.println(result);

        result--;
        // result is now 0
        System.out.println(result);

        result++;
        // result is now 1
        System.out.println(result);

        result = -result;
        // result is now -1
        System.out.println(result);

        boolean success = false;
        // false
        System.out.println(success);
        // true
        System.out.println(!success);
    }

}
```

递增/递减运算符可以之前（前缀）或（后缀）操作数后应用。该代码 `result++;` 和 `++result;` 两个的 result 都被加一。唯一的区别是，该前缀版本 `++result;` 增递增了，而后缀版本 `result++;` 的计算结果为原始值。下面是 PrePostDemo 的示例，说明了两者的区别：

```java
class PrePostDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int i = 3;
        i++;
        // prints 4
        System.out.println(i);
        ++i;
        // prints 5
        System.out.println(i);
        // prints 6
        System.out.println(++i);
        // prints 6
        System.out.println(i++);
        // prints 7
        System.out.println(i);
    }
}
```

等价和关系运算符

等价和关系运算符包括

运算符 | 描述 |:----:| ----| |== | 相等（equal to）| |!= | 不相等（not equal to）| |> | 大于（greater than）| |>= | 大于等于（greater than or equal to）| |< | 小于（less than）| |<= | 小于等于（less than or equal to）|

ComparisonDemo 对比的例子：

```
class ComparisonDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if (value1 == value2)
            System.out.println("value1 == value2");
        if (value1 != value2)
            System.out.println("value1 != value2");
        if (value1 > value2)
            System.out.println("value1 > value2");
        if (value1 < value2)
            System.out.println("value1 < value2");
        if (value1 <= value2)
            System.out.println("value1 <= value2");
    }
}
```

输出为：

```
value1 != value2
value1 < value2
value1 <= value2
```

条件运算符

条件运算符包括：

运算符 | 描述 |:----:| ----| | `&&` | 条件与（Conditional-AND）| | `||` | 条件或（Conditional-OR）| | `?:` | 三元运算符（ternary operator）|

条件与、条件或的运算符例子 ConditionalDemo1：

```
class ConditionalDemo1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        if ((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if ((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```

输出：

value1 is 1 AND value2 is 2 value1 is 1 OR value2 is 1

下面是一个三元运算符的例子,类似与 if-then-else 语句，见 ConditionalDemo2 ：

```
class ConditionalDemo2 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = true;
        result = someCondition ? value1 : value2;

        System.out.println(result);
    }
}
```

instanceof 运算符

instanceof 用于匹配判断对象的类型。可以用它来测试对象是否是类的一个实例，子类的实例，或者是实现了一个特定接口的类的实例。见例子 InstanceofDemo,父类是 Parent ，接口是 MyInterface ，子类是 Child 继承了父类并实现了接口。

```
class InstanceofDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {

        // Must qualify the allocation with an enclosing instance of type InstanceofDe
mo
        Parent obj1 = new InstanceofDemo().new Parent();
        Parent obj2 = new InstanceofDemo().new Child();

        System.out.println("obj1 instanceof Parent: "
            + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: "
            + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: "
            + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: "
            + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: "
            + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: "
            + (obj2 instanceof MyInterface));
    }

    class Parent {}
    class Child extends Parent implements MyInterface {}
    interface MyInterface {}
}
```

输出为：

```
obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true
```

注：null 不是任何类的实例

位运算和位移运算符

位运算和位移运算符适用于整型。

位运算符

运算符 | 描述 |:----:| ----| | `&` | 与| | `|` | 或| | `^` | 异或| | `~` | 非（把0变成1，把1变成0）|

BitDemo 例子：

```
class BitDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int bitmask = 0x000F;
        int val = 0x2222;
        // prints "2"
        System.out.println(val & bitmask);
    }
}
```

位移运算符

首先我们先阐述一下符号位的概念:

- 符号位：是数的最后一位，不用来计算的；
- 当符号位为0时，值为正数；当符号位为1时，值为负数；
- 无符号位时为正数，有符号位时为正数或者负数；

运算符 | 描述 |:----:| ----| | `<<` | 左移| | `>>` | 右移| | `>>>` | 右移（补零）|

左移（<<） 运算形式：值 << 位数

右移（>>） 运算形式：值 >> 位数

移动后，左移、右移都会保留符号位！

右移（补零），移动后，不保留符号位，永远为正数，因为其符号位总是被补零；

源码

本章例子的源码，可以在 `com.waylau.essentialjava.operator` 包下找到。

表达式、语句和块

运算符用于计算构建成了表达式(expressions)，而表达式是语句(statements)的核心组成，而语句是组织形式为块(blocks)。

表达式

表达式是由变量、运算符以及方法调用所构成的结构，如下：

```
int cadence = 0;
anArray[0] = 100;
System.out.println("Element 1 at index 0: " + anArray[0]);

int result = 1 + 2; // result is now 3
if (value1 == value2)
    System.out.println("value1 == value2");
```

表达式返回的数据类型取决于表达式中的元素。表达式 `cadence = 0` 返回一个int，因为赋值运算符将返回相同的数据类型作为其左侧操作数的值;在这种情况下，cadence 是一个 int。

下面是一个复合表达式：

```
1 * 2 * 3
```

表达式应该尽量避免歧义，比如：

```
x + y / 100
```

有歧义，推荐写成 `(x + y) / 100` 或 `x + (y / 100)` 。

语句

语句相当于自然语言中的句子。一条语句就是一个执行单元。用分号（;）结束一条语句。下面是表达式语句（expression statements），包括：

- 赋值表达式（Assignment expressions）
- ++ 或者 --（Any use of ++ or --）
- 方法调用（Method invocations—）
- 对象创建（Object creation expressions）

下面是表达式语句的例子

```
// assignment statement
aValue = 8933.234;
// increment statement
aValue++;
// method invocation statement
System.out.println("Hello World!");
// object creation statement
Bicycle myBike = new Bicycle();
```

除了表达式语句，其他的还有声明语句（declaration statements）：

```
// declaration statement
double aValue = 8933.234;
```

以及控制流程语句（control flow statements）：

```
if (isMoving)
        currentSpeed--;
```

块

块是一组零个或多个成对大括号之间的语句，并可以在任何地方允许使用一个单独的语句。下面的 BlockDemo 例子：

```
class BlockDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block one
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

源码

本章例子的源码，可以在 `com.waylau.essentialjava.expression` 包下找到。

控制流程语句

控制流程语句用于控制程序按照一定流程来执行。

if-then

它告诉你要只有 if 后面是 true 时才执行特定的代码。

```
void applyBrakes() {
    // the "if" clause: bicycle must be moving
    if (isMoving){
        // the "then" clause: decrease current speed
        currentSpeed--;
    }
}
```

如果 if 后面是 false, 则跳到 if-then 语句后面。语句可以省略中括号，但在编码规范里面不推荐使用，如：

```
void applyBrakes() {
    // same as above, but without braces
    if (isMoving)
        currentSpeed--;
}
```

if-then-else

该语句是在 if 后面是 false 时，提供了第二个执行路径。

```
void applyBrakes() {
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

下面是一个完整的例子：

```
class IfElseDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
    }
}
```

输出为：`Grade = C`

switch

switch 语句可以有许多可能的执行路径。可以使用 byte, short, char, 和 int 基本数据类型，也可以是枚举类型（enumerated types）、String 以及少量的原始类型的包装类 Character, Byte, Short, 和 Integer。

下面是一个 SwitchDemo 例子：

```
class SwitchDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int month = 8;
        String monthString;
        switch (month) {
        case 1:
            monthString = "January";
            break;
        case 2:
            monthString = "February";
```

```
            break;
        case 3:
            monthString = "March";
            break;
        case 4:
            monthString = "April";
            break;
        case 5:
            monthString = "May";
            break;
        case 6:
            monthString = "June";
            break;
        case 7:
            monthString = "July";
            break;
        case 8:
            monthString = "August";
            break;
        case 9:
            monthString = "September";
            break;
        case 10:
            monthString = "October";
            break;
        case 11:
            monthString = "November";
            break;
        case 12:
            monthString = "December";
            break;
        default:
            monthString = "Invalid month";
            break;
        }
        System.out.println(monthString);
    }
}
```

break 语句是为了防止 fall through。

```
class SwitchDemoFallThrough {

    /**
     * @param args
     */
    public static void main(String[] args) {
        java.util.ArrayList<String> futureMonths = new java.util.ArrayList<String>();

        int month = 8;

        switch (month) {
        case 1:
            futureMonths.add("January");
        case 2:
            futureMonths.add("February");
        case 3:
            futureMonths.add("March");
        case 4:
            futureMonths.add("April");
        case 5:
            futureMonths.add("May");
        case 6:
            futureMonths.add("June");
        case 7:
            futureMonths.add("July");
        case 8:
            futureMonths.add("August");
        case 9:
            futureMonths.add("September");
        case 10:
            futureMonths.add("October");
        case 11:
            futureMonths.add("November");
        case 12:
            futureMonths.add("December");
            break;
        default:
            break;
        }

        if (futureMonths.isEmpty()) {
            System.out.println("Invalid month number");
        } else {
            for (String monthName : futureMonths) {
                System.out.println(monthName);
            }
        }
    }

}
```

输出为：

```
August
September
October
November
December
```

技术上来说，最后一个 break 并不是必须，因为流程跳出 switch 语句。但仍然推荐使用 break ，主要修改代码就会更加简单和防止出错。default 处理了所有不明确值的情况。

下面例子展示了一个局域多个 case 的情况，

```
class SwitchDemo2 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            numDays = 31;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            numDays = 30;
            break;
        case 2:
            if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))
                numDays = 29;
            else
                numDays = 28;
            break;
        default:
            System.out.println("Invalid month.");
            break;
        }
        System.out.println("Number of Days = " + numDays);
    }
}
```

输出为：`Number of Days = 29`

使用 String

Java SE 7 开始，可以在 switch 语句里面使用 String,下面是一个例子

```
class StringSwitchDemo {

    public static int getMonthNumber(String month) {
```

```
        int monthNumber = 0;

        if (month == null) {
            return monthNumber;
        }

        switch (month.toLowerCase()) {
        case "january":
            monthNumber = 1;
            break;
        case "february":
            monthNumber = 2;
            break;
        case "march":
            monthNumber = 3;
            break;
        case "april":
            monthNumber = 4;
            break;
        case "may":
            monthNumber = 5;
            break;
        case "june":
            monthNumber = 6;
            break;
        case "july":
            monthNumber = 7;
            break;
        case "august":
            monthNumber = 8;
            break;
        case "september":
            monthNumber = 9;
            break;
        case "october":
            monthNumber = 10;
            break;
        case "november":
            monthNumber = 11;
            break;
        case "december":
            monthNumber = 12;
            break;
        default:
            monthNumber = 0;
            break;
        }

        return monthNumber;
    }

    public static void main(String[] args) {
```

```
        String month = "August";

        int returnedMonthNumber = StringSwitchDemo.getMonthNumber(month);

        if (returnedMonthNumber == 0) {
            System.out.println("Invalid month");
        } else {
            System.out.println(returnedMonthNumber);
        }
    }
}
```

输出为：8

注：switch 语句表达式中不能有 null。

while

while 语句在判断条件是 true 时执行语句块。语法如下：

```
while (expression) {
     statement(s)
}
```

while 语句计算的表达式，必须返回 boolean 值。如果表达式计算为 true，while 语句执行 while 块的所有语句。while 语句继续测试表达式，然后执行它的块，直到表达式计算为 false。完整的例子：

```
class WhileDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }
    }
}
```

用 while 语句实现一个无限循环：

```
while (true){
    // your code goes here
}
```

do-while

语法如下：

```
do {
     statement(s)
} while (expression);
```

do-while 语句和 while 语句的区别是，do-while 计算它的表达式是在循环的底部，而不是顶部。所以，do 块的语句，至少会执行一次，如 DoWhileDemo 程序所示：

```
class DoWhileDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count < 11);
    }
}
```

输出为：

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

for

for 语句提供了一个紧凑的方式来遍历一个范围值。程序经常引用为"for 循环"，因为它反复循环，直到满足特定的条件。for 语句的通常形式，表述如下：

```
for (initialization; termination;
     increment) {
    statement(s)
}
```

使 for 语句时要注意：

- initialization 初始化循环；它执行一次作为循环的开始。
- 当 termination 计算为 false，循环结束。
- increment 会在循环的每次迭代执行；该表达式可以接受递增或者递减的值

```
class ForDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
         }
    }

}
```

输出为：

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

注意：代码在 initialization 声明变量。该变量的存活范围，从它的声明到 for 语句的块的结束。所以，它可以用在 termination 和 increment。如果控制 for 语句的变量，不需要在循环外部使用，最好是在 initialization 声明。经常使用 i,j,k 经常用来控制 for 循环。在 initialization 声明他们，可以限制他们的生命周期，减少错误。

for 循环的三个表达式都是可选的，一个无限循环，可以这么写：

```
// infinite loop
for ( ; ; ) {

    // your code goes here
}
```

for 语句还可以用来迭代 集合（Collections） 和 数组（arrays），这个形式有时被称为增强的 for 语句（ enhanced for ），可以用来让你的循环更加紧凑，易于阅读。为了说明这一点，考虑下面的数组：

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

使用 增强的 for 语句来循环数组

```
class EnhancedForDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int[] numbers =
            {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: " + item);
        }
    }

}
```

输出：

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

尽可能使用这种形式的 for 替代传统的 for 形式。

break

break 语句有两种形式：标签和非标签。在前面的 switch 语句，看到的 break 语句就是非标签形式。可以使用非标签 break 用来结束 for，while，do-while 循环，如下面的 BreakDemo 程序：

```
class BreakDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
        int searchfor = 12;

        int i;
        boolean foundIt = false;

        for (i = 0; i < arrayOfInts.length; i++) {
            if (arrayOfInts[i] == searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }

}
```

这个程序在数组终查找数字12。break 语句，当找到值时，结束 for 循环。控制流就跳转到 for 循环后面的语句。程序输出是：

```
Found 12 at index 4
```

无标签 break 语句结束最里面的 switch，for，while，do-while 语句。而标签break 结束最外面的语句。接下来的程序，BreakWithLabelDemo，类似前面的程序，但使用嵌套循环在二维数组里寻找一个值。但值找到后，标签 break 语句结束最外面的 for 循环(标签为"search"):

```
class BreakWithLabelDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        int[][] arrayOfInts = { { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127,
77, 955 } };
        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

        search: for (i = 0; i < arrayOfInts.length; i++) {
            for (j = 0; j < arrayOfInts[i].length; j++) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}
```

程序输出是：

```
Found 12 at 1, 0
```

break 语句结束标签语句，它不是传送控制流到标签处。控制流传送到紧随标记（终止）声明。

注： Java 没有类似于 C 语言的 goto 语句，但带标签的 break 语句，实现了类似的效果。

continue

continue 语句忽略 for，while，do-while 的当前迭代。非标签模式，忽略最里面的循环体，然后计算循环控制的 boolean 表达式。接下来的程序，ContinueDemo，通过一个字符串的步骤，计算字母“p”出现的次数。如果当前字符不是 p，continue 语句跳过循环的其他代码，然

后处理下一个字符。如果当前字符是 p，程序自增字符数。

```
class ContinueDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String searchMe = "peter piper picked a " + "peck of pickled peppers";
        int max = searchMe.length();
        int numPs = 0;

        for (int i = 0; i < max; i++) {
            // interested only in p's
            if (searchMe.charAt(i) != 'p')
                continue;

            // process p's
            numPs++;
        }
        System.out.println("Found " + numPs + " p's in the string.");
    }

}
```

程序输出：

```
Found 9 p's in the string
```

为了更清晰看效果，尝试去掉 continue 语句，重新编译。再跑程序，count 将是错误的，输出是 35，而不是 9.

带标签的 continue 语句忽略标签标记的外层循环的当前迭代。下面的程序例子，ContinueWithLabelDemo，使用嵌套循环在字符传的字串中搜索字串。需要两个嵌套循环：一个迭代字串，一个迭代正在被搜索的字串。下面的程序ContinueWithLabelDemo，使用 continue 的标签形式，忽略最外层的循环。

```
class ContinueWithLabelDemo {

    /**
     * @param args
     */
    public static void main(String[] args) {
        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;

        int max = searchMe.length() - substring.length();

        test: for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
            break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }

}
```

这里是程序输出：

```
Found it
```

return

最后的分支语句是 return 语句。return 语句从当前方法退出，控制流返回到方法调用处。return 语句有两种形式：一个是返回值，一个是不返回值。为了返回一个值，简单在 return 关键字后面把值放进去(或者放一个表达式计算)

```
return ++count;
```

return 的值的数据类型，必须和方法声明的返回值的类型符合。当方法声明为 void，使用下面形式的 return 不需要返回值。

```
return;
```

源码

本章例子的源码，可以在 `com.waylau.essentialjava.flow` 包下找到。

类和对象

枚举类型（Enum Type）

枚举类型是一种特殊的数据类型，使一个变量是一组预定义的常量。变量必须等于已预先定义的值之一。常见的例子包括罗盘方向（ NORTH, SOUTH, EAST 和 WEST）和星期几。

使用关键字 enum ，下面是一个星期几的枚举列子:

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

使用枚举类型，需要一组固定的常数。这包括自然枚举类型，如在我们的太阳系的行星，菜单上的选项，命令行标志，等等。

下面是一些代码，展示如何使用上面定义的 Day 枚举：

```
class EnumTest {

    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest sixthDay = new EnumTest(Day.SATURDAY);
        sixthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

输出为：

```
Mondays are bad.
Midweek days are so-so.
Fridays are better.
Weekends are best.
Weekends are best.
```

下面是一个 Planet 示例，展示了 枚举值的 for-each 遍历：

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass() { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Planet <earth_weight>");
            System.exit(-1);
        }
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
}
```

在命令行，输入参数为 175 时， 输出如下：

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
```

注解（Annotations）

注解为程序提供元数据（metadata）.元数据又称中介数据、中继数据，为描述数据的数据（data about data），主要是描述数据属性（property）的信息。它不会影响程序的编译方式，也不会影响最终的编译结果。

注解有如下的使用场景：

- 编译器信息— 编译器用注解检测到错误或抑制警告。
- 编译时和部署时的处理 — 软件工具可以处理注释的信息来生成代码，XML文件，等等。
- 运行时处理 — 有些注解是在运行时进行检查.

注解的格式

注解的格式的通常拥有键/值对，其键就是方法名。 格式如下：

```
@Entity
```

符号 `@` 告诉编译器这是个注解。

注解可以包含有名字或者没有名字的元素（elements），如：

```
@Author(
   name = "Benjamin Franklin",
   date = "3/27/2003"
)
class MyClass() { ... }
```

或者

```
@SuppressWarnings(value = "unchecked")
void myMethod() { ... }
```

当只有一个元素名字是 value 时，该名字可以省略，如：

```
@SuppressWarnings("unchecked")
void myMethod() { ... }
```

若注解没有元素，则连圆括号都可以省略。

同一个声明可以用多个注解：

```
@Author(name = "Jane Doe")
@EBook
class MyClass { ... }
```

若注解包含相同的类型，则被称为重复注解(repeating annotation)：

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClass { ... }
```

重复注解是 Java SE 8 里面支持的。

注解使用的地方

注解可以应用到程序声明的类，字段，方法，和其他程序元素。当在一个声明中使用，按照惯例，每个注解经常会出现在它自己的行。

Java SE8 开始，注解也可以应用于类型使用（type use），称为类型注解（type annotation）。这里有些例子：

- 类实例创建表达式

```
new @Interned MyObject()
```

- 类型投射

```
myString = (@NonNull String) str;
```

- 实现条款

```
class UnmodifiableList<T> implements
        @Readonly List<@Readonly T> { ... }
```

- 抛出异常声明

```
void monitorTemperature() throws
        @Critical TemperatureException { ... }
```

声明一个注解类型

许多注解取代了本来已经在代码中的注释。

假设传统的软件组在每个类的类体的开始，使用注释提供了重要的信息：

```
public class Generation3List extends Generation2List {

   // Author: John Doe
   // Date: 3/17/2002
   // Current revision: 6
   // Last modified: 4/12/2004
   // By: Jane Doe
   // Reviewers: Alice, Bill, Cindy

   // class code goes here

}
```

使用注解提供一样的元数据，首先要声明一个注解类型，语法是：

```
@interface ClassPreamble {
   String author();
   String date();
   int currentRevision() default 1;
   String lastModified() default "N/A";
   String lastModifiedBy() default "N/A";
   // Note use of array
   String[] reviewers();
}
```

注解的声明，就像在 interface 声明前面添加一个 `@` 字符(`@` 是AT,即 Annotation Type)。注解类型，其实是接口的一种特殊形式，后面会讲到。就目前而言，你不需要了解。

注解的声明的正文，包括注解元素的声明，看起来很像方法。注意，这里可以定义可选的默认值。

一旦注解定义好了，就可以在使用注解时，填充注解的值，就像这样：

```
@ClassPreamble (
   author = "John Doe",
   date = "3/17/2002",
   currentRevision = 6,
   lastModified = "4/12/2004",
   lastModifiedBy = "Jane Doe",
   // Note array notation
   reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {

// class code goes here

}
```

注：要让 `@ClassPreamble` 的信息出现在 Javadoc 生成的文档，必须使用 `@Documented` 注解定义 `@ClassPreamble`

```
// import this to use @Documented
import java.lang.annotation.*;

@Documented
@interface ClassPreamble {

   // Annotation element definitions

}
```

预定义注解的类型

有这么几种注解类型预定义在 Java SE API 了。一些注解类型是供 Java 编译器使用，一些是供其他注解使用。

Java 语言使用的注解

定义在 java.lang 中的是 `@Deprecated`，`@Override`，和 `@SuppressWarnings`

`@Deprecated` 注解指示，标识的元素是废弃的(deprecated)，不应该再使用。编译器会在任何使用到 `@Deprecated` 的类，方法，字段的程序时产生警告。当元素是废弃的，它也应该使用 Javadoc 的 `@deprecated` 标识文档化，如下面的例子。两个Javadoc 注释和注解中的“@”符号的使用不是巧合 - 它们是相关的概念上。另外，请注意Javadoc标记开始用小写字母“d”和注解开始以大写字母“D”。

```
// Javadoc comment follows
    /**
     * @deprecated
     * explanation of why it was deprecated
     */
    @Deprecated
    static void deprecatedMethod() { }
}
```

`@Override` 注解通知编译器，覆盖父类声明的元素。

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

虽然不要求在覆盖方法时，必须使用注解，但是它可以避免错误。如果一个方法标记为 `@Override` ，但是无法正确覆盖父类的任何方法，编译器会产生错误。

`@SuppressWarnings` 告诉编译器，抑制正常情况下会产生的特定的警告。下面的例子，一个废弃的方法被使用，编译器正常会产生警告，而这个情况下，这个注解导致警告会被抑制。

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    // deprecation warning
    // - suppressed
    objectOne.deprecatedMethod();
}
```

每个编译器的警告属于一个类别。Java 语言规范有两个类别："deprecation" 和"unchecked"。"unchecked" 会在使用以前的写的泛型的遗留代码进行交互时，产生警告。抑制更多类别的警告，使用下面的语法：

```
@SuppressWarnings({"unchecked", "deprecation"})
```

`@SafeVarargs` 注解，当应用于方法或构造，断言代码不对其可变参数（varargs）的参数进行潜在的不安全操作。当使用这个注释类型时，与可变参数相关未检查警告被抑制。

`@FunctionalInterface` 是在 Java SE 8 中引入，由 Java 语言规范定义的那样，表示该类型声明意在成为功能性的接口。

注解应用于其他注解

注解应用于其他注解称为元注解（ meta-annotations）。java.lang.annotation 中定义了多种元注解。

`@Retention` 注解指定了标记的注解如何存储：

- RetentionPolicy.SOURCE - 该标记注解只保留在源码级，在编译阶段丢弃。这些注解在编译结束之后就不再有任何意义，所以它们不会写入字节码。`@Override`、`@SuppressWarnings` 都属于这类注解。
- RetentionPolicy.CLASS - 该标记注释是由编译器在编译时保留， 在类加载的时候丢弃。在字节码文件的处理中有用。注解默认使用这种方式。
- RetentionPolicy.RUNTIME - 该标记注解由JVM保留，因此可以使用在运行时环境。因此可以使用反射机制读取该注解的信息。我们自定义的注解通常使用这种方式。

`@Documented` 注释表明，只要指定哪些元素应该使用 Javadoc 工具。 （默认情况下，注解不包括在 Javadoc 中。）有关详细信息，请参阅的 Javadoc 工具页面。

`@Target` 用于标记其他注解，限制什么样的 Java 元素的注解可以应用到。`@Target` 注解指定以下元素类型作为其值之一：

- ElementType.ANNOTATION_TYPE 可以应用于注释类型。
- ElementType.CONSTRUCTOR 可以应用于构造体。
- ElementType.FIELD 可以应用于一个字段或属性。
- ElementType.LOCAL_VARIABLE 可以应用到局部变量。
- ElementType.METHOD 可以应用于一方法级注释。
- ElementType.PACKAGE 可以应用到一个包声明。
- ElementType.PARAMETER 可以应用于方法的参数。
- ElementType.TYPE 可以应用于类的任意元素。

`@Inherited` 指示注释类型可以从超类继承。（默认不是 true）。当用户查询注释类型,类没有这种类型注释，此时从这个类的父类中查询注释类型。这个注释只适用于类的声明。

`@Repeatable` 注解，在 Java SE 8 中引入的，表示该标记的注解可以多次应用到同一声明或类型使用。欲了解更多信息，请参阅重复注解。

类型注解以及可拔插的类型系统

Java SE8 之前，注解只能用于声明，从 Java SE8 开始，注解也可以应用于类型使用（type use），称为类型注解（type annotation）。意味着，注解可以使用在任何使用的类型。

类型注解为 Java 程序提供了更强的类型检查分析。Java SE 8 版本不提供类型检查的框架，但它可以让你自己写（或下载）类型检查框架，该框架实现了与 Java 编译器一起使用的一个或多个可插拔模块。

例如，要确保在你的程序中一个特定变量从未被分配到 null ,从而避免引发 NullPointerException 异常。您可以编写自定义插件来检查这一点。然后，您可以修改代码以注明这个特定变量，以表明它是永远不会分配给 null。变量声明可能是这样的：

```
@NonNull String str;
```

当您编译代码，包括在命令行中的 NonNull 模块，如果它检测到潜在的问题，编译器输出警告，让您可以修改代码以避免错误。在更正代码后，消除所有警告，当程序运行时不会发生此特定错误。

您可以使用多个类型检查的模块，每个模块检查不同类型的错误。通过这种方式，你可以建立在 Java 类型系统之上，随时随地添加您想要的特定检查。

通过明智地使用类型注解和可插拔的类型检查器，你写的代码，将更强大，更不易出错。

在很多情况下，你不必写自己的类型检查模块。第三方组织已经在做这个工作了。例如，华盛顿大学(the University of Washington)创建的 Checker Framework 。该框架包括一个 NonNull 模块，以及一个正则表达式模块和互斥锁模块。欲了解更多信息，请参见 Checker Framework。

重复注解

若注解包含相同的类型，则被称为重复注解(repeating annotation)，这个是 Java SE 8 之后所支持的。

比如，你正在编写的代码使用计时器服务，使您能够在特定的时间或在某个计划，类似于 UNIX cron 服务运行的方法。现在，你要设置一个计时器，在下午 11:00 运行的方法，doPeriodicCleanup，在每月和每周五的最后一天要设置定时运行，创建一个 `@Schedule` 注释，并两次将其应用到了 doPeriodicCleanup 方法。在第一次使用指定月的最后一天和第二指定星期五在下午11点，使用如下：

```
@Schedule(dayOfMonth="last")
@Schedule(dayOfWeek="Fri", hour="23")
public void doPeriodicCleanup() { ... }
```

上面的示例是将注解应用在方法上。你可以在任何使用标准的注解地方使用重复注解。例如，你有一个类来处理未授权的访问异常。有一个 `@Alert` 注解的类标注为管理人员和另一个用于管理员：

```
@Alert(role="Manager")
@Alert(role="Administrator")
public class UnauthorizedAccessException extends SecurityException { ... }
```

由于兼容性的原因，重复的注释被存储在一个由 Java 编译器自动产生的容器注解（container annotation）里。为了使编译器要做到这一点，你的代码里两个声明都需要。

第一步：声明一个重复注解

重复注解用 @Repeatable 元注解标记。下面例子定义一个自定义的 @Schedule 重复注解：

```
import java.lang.annotation.Repeatable;

@Repeatable(Schedules.class)
public @interface Schedule {
  String dayOfMonth() default "first";
  String dayOfWeek() default "Mon";
  int hour() default 12;
}
```

`@Repeatable` 元注解的值是由 Java 编译器生成存储重复注解的容器注解的类型。在本例中，容器注解的类型是 Schedules，所以重复注解 `@Schedule` 被存储在 `@Schedules` 注解中。

应用相同注解到声明但没有首先声明它是可重复的，则在编译时会出错。

步骤2：声明容器注解类型

容器注解类型必须有数组类型的元素 value,而数组类型的组件类型必须是重复注解类型，示例如下：

```
public @interface Schedules {
    Schedule[] value();
}
```

检索注解

反射 API 有几种方法可用于检索注解。返回单个注解的方法的行为，如 AnnotatedElement.getAnnotationByType(Class)，如果所请求的类型的注解类型只存在一个则仅返回一个注解，则该行为是没有改变的。如果有多个请求类型的注解类型存在，则可以通过先得到他们的容器注解从而获取它们。这种方式下，传统代码继续工作。其他方法是在 Java SE 8 中，通过容器注释扫描到一次返回多个注解，如 AnnotatedElement.getAnnotations(Class)。见AnnotatedElement 类的规范，查看所有的可用方法的信息。

设计考虑

当设计一个注解类型，你必须考虑到该类型的注解的基数(cardinality)。现在可以使用一个注释零次，一次，或者，如果注解的类型被标以 `@Repeatable` ，则不止一次。另外，也可以通过使用 `@Target` 元注解来限制注解类型在哪里使用。例如，您可以创建一个只能在方法和字段使用可重复的注解类型。精心设计的注解类型是非常重要的，要确保使用注解的程序员感觉越灵活和强大越好。

示例

如何定义注解

我们自定义了一个注解 MyAnnotation，用来标识我们是什么公司:

```
@Documented
@Retention(RUNTIME)
public @interface MyAnnotation {
    String company() default "waylau.com";
}
```

该注解只有一个方法声明 company()，默认值是字符串“waylau.com”。

如何使用注解

下面演示下如何用这个注解。

我们在测试类 AnnotationTest 的方法上加上了我们的注解，并设了值“www.waylau.com”：

```
class AnnotationTest {

    @MyAnnotation(company="https://waylau.com")
    public void execute(){
        System.out.println("do something~");
    }
}
```

如果获取注解的信息

通过反射机制，我们可以获取到注解的信息：

```
AnnotationTest test = new AnnotationTest();

test.execute();

// 获取 AnnotationTest 的Class实例
Class<AnnotationTest> c = AnnotationTest.class;

// 获取需要处理的方法Method实例
Method method = c.getMethod("execute", new Class[]{});

// 判断该方法是否包含 MyAnnotation 注解
if(method.isAnnotationPresent(MyAnnotation.class)){

    // 获取该方法的 MyAnnotation 注解实例
    MyAnnotation myAnnotation = method.getAnnotation(MyAnnotation.class);

    // 执行该方法
    method.invoke(test, new Object[]{});

    // 获取 myAnnotation 的属性值
    String company = myAnnotation.company();
    System.out.println(company);
}

// 获取方法上的所有注解
Annotation[] annotations = method.getAnnotations();
    for(Annotation annotation : annotations){
        System.out.println(annotation);
    }
}
```

执行，正常情况下能看到如下打印信息：



泛型（Generics）

泛型通过在编译时检测到更多的代码 bug 从而使你的代码更加稳定。

泛型的作用

概括地说，泛型支持类型（类和接口）在定义类，接口和方法时作为参数。就像在方法声明中使用的形式参数（formal parameters），类型参数提供了一种输入可以不同但代码可以重用的方式。所不同的是，形式参数的输入是值，类型参数输入的是类型参数。

使用泛型对比非泛型代码有很多好处：

- 在编译时更强的类型检查。

如果代码违反了类型安全，Java 编译器将针对泛型和问题错误采用强大的类型检查。修正编译时的错误比修正运行时的错误更加容易。

- 消除了强制类型转换。

没有泛型的代码片需要强制转化：

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

当重新编写使用泛型，代码不需要强转：

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- 使编程人员能够实现通用算法。

通过使用泛型，程序员可以实现工作在不同类型集合的通用算法，并且是可定制，类型安全，易于阅读。

泛型类型（Generic Types）

泛型类型是参数化类型的泛型类或接口。下面是一个 Box 类例子来说明这个概念。

一个简单的 Box 类

```
public class Box {
    private Object object;

    public void set(Object object) {
        this.object = object;
    }

    public Object get() {
        return object;
    }
}
```

由于它的方法接受或返回一个 Object，你可以自由地传入任何你想要的类型，只要它不是原始的类型之一。在编译时，没有办法验证如何使用这个类。代码的一部分可以设置 Integer 并期望得到 Integer ，而代码的另一部分可能会由于错误地传递一个String ，而导致运行错误。

一个泛型版本的 Box 类

泛型类定义语法如下：

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

类型参数部分用 `<>` 包裹，制定了类型参数或称为类型变量（type parameters or type variables) T1, T2, ..., 直到 Tn.

下面是代码：

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

}
```

主要，所有的 Object 被 T 代替了。类型变量可以是非基本类型的的任意类型，任意的类、接口、数组或其他类型变量。

这个技术同样适用于泛型接口的创建。

类型参数命名规范

按照惯例，类型参数名称是单个大写字母，用来区别普通的类或接口名称。

常用的类型参数名称如下：

```
E - Element (由 Java 集合框架广泛使用)
K - Key
N - Number
T - Type
V - Value
S,U,V 等. - 第二种、第三种、第四种类型
```

调用和实例化一个泛型

从代码中引用泛型 Box 类，则必须执行一个泛型调用(generic type invocation)，用具体的值，比如 Integer 取代 T ：

```
Box<Integer> integerBox;
```

泛型调用与普通的方法调用类似，所不同的是传递参数是类型参数（type argument ），本例就是传递 Integer 到 Box 类：

Type Parameter 和 *Type Argument* 区别

编码时，提供 *type argument* 的一个原因是为了创建 参数化类型。因此，`Foo<T>` 中的 *T* 是一个 *type parameter*， 而 `Foo<String>` 中的 *String* 是一个 *type argument*

与其他变量声明类似，代码实际上没有创建一个新的 Box 对象。它只是声明integerBox 在读到 `Box<Integer>` 时，保存一个“Integer 的 Box”的引用。

泛型的调用通常被称为一个参数化类型（parameterized type）。

实例化类，使用 new 关键字：

```
Box<Integer> integerBox = new Box<Integer>();
```

菱形（**Diamond**）

Java SE 7 开始泛型可以使用空的类型参数集 `<>` ，只要编译器能够确定，或推断，该类型参数所需的类型参数。这对尖括号 `<>` ，被非正式地称为“菱形（diamond）”。例如：

```
Box<Integer> integerBox = new Box<>();
```

多类型参数

下面是一个泛型 Pair 接口和一个泛型 OrderedPair ：

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey()    { return key; }
    public V getValue() { return value; }
}
```

创建两个 OrderedPair 实例：

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

代码 `new OrderedPair<String, Integer>` ，实例 K 作为一个 String 和 V 为 Integer。因此，OrderedPair 的构造函数的参数类型是 String 和 Integer。由于自动装箱（autoboxing），可以有效的传递一个 String 和 int 到这个类。

可以使用菱形（diamond）来简化代码：

```
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
```

参数化类型

您也可以用 参数化类型（例如， `List<String>` 的）来替换类型参数（即 K 或 V ）。例如，使用 `OrderedPair<K，V>` 例如：

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...
));
```

原生类型（Raw Types）

原生类型是没有类型参数(type arguments)的泛型类和泛型接口，如泛型 Box 类;

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

为了创建

关键字

下面是 Java 里面的关键字。不能使用以下任一作为您的程序标识符。关键字 const 和 goto 语句被保留，即使他们目前尚未使用。true, false, 和 null 似乎是关键字，但它们实际上是字面值;你不能使用它们作为你的程序标识符。

```
abstract    continue    for    new    switch
assert***    default    goto*    package    synchronized
boolean    do    if    private    this
break    double    implements    protected    throw
byte    else    import    public    throws
case    enum****    instanceof    return    transient
catch    extends    int    short    try
char    final    interface    static    void
class    finally    long    strictfp**    volatile
const*    float    native    super    while
```

其中: `*` 表示未使用， `**` 表示是 1.2版本加入， `***` 表示 1.4版本加入， `****` 表示5.0版本加入

IO

本章交要讲解基本的 I/O 。它首先集中在 “I/O流”（I/O Streams），一个强大的概念用于简化 I/O 操作。本文还讲解了序列化，它可以让程序将整个对象转出为流，然后再从流读回来。随后介绍文件 I/O 和文件系统的操作，其中包括了随机访问文件。

大多数涵盖 I/O流 的类都在 `java.io` 包。大多数涵盖文件 I/O 的类都在 `java.nio.file` 包。

I/O 流

字节流(Byte Streams)

字节流处理原始的二进制数据 I/O。输入输出的是8位字节，相关的类为 InputStream 和 OutputStream.

字节流的类有许多。为了演示字节流的工作，我们将重点放在文件 I/O字节流 FileInputStream 和 FileOutputStream 上。其他种类的字节流用法类似，主要区别在于它们构造的方式，大家可以举一反三。

用法

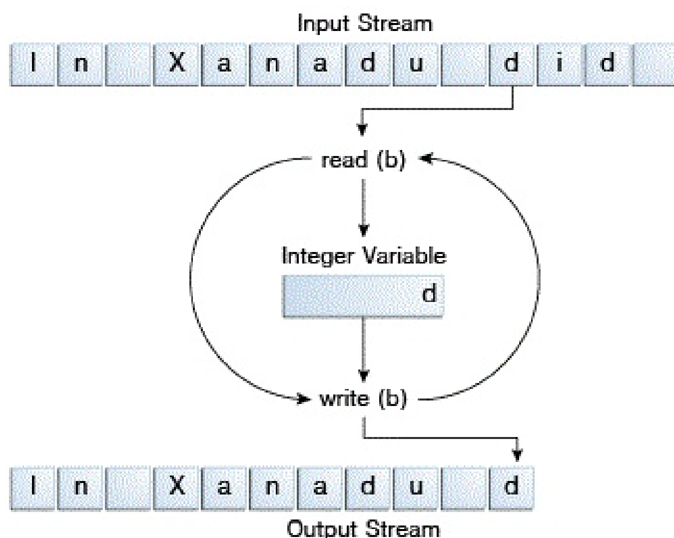
下面一例子 CopyBytes， 从 xanadu.txt 文件复制到 outagain.txt，每次只复制一个字节：

```
public class CopyBytes {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("resources/xanadu.txt");
            out = new FileOutputStream("resources/outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes 花费其大部分时间在简单的循环里面，从输入流每次读取一个字节到输出流，如图所示：



记得始终关闭流

不再需要一个流记得要关闭它，这点很重要。所以，CopyBytes 使用 finally 块来保证即使发生错误两个流还是能被关闭。这种做法有助于避免严重的资源泄漏。

一个可能的错误是，CopyBytes 无法打开一个或两个文件。当发生这种情况，对应解决方案是判断该文件的流是否是其初始 null 值。这就是为什么 CopyBytes 可以确保每个流变量在调用前都包含了一个对象的引用。

何时不使用字节流

CopyBytes 似乎是一个正常的程序，但它实际上代表了一种低级别的 I/O，你应该避免。因为 xanadu.txt 包含字符数据时，最好的方法是使用字符流，下文会有讨论。字节流应只用于最原始的 I/O。所有其他流类型是建立在字节流之上的。

字符流(Character Streams)

字符流处理字符数据的 I/O，自动处理与本地字符集转化。

Java 平台存储字符值使用 Unicode 约定。字符流 I/O 会自动将这个内部格式与本地字符集进行转换。在西方的语言环境中，本地字符集通常是 ASCII 的8位超集。

对于大多数应用，字符流的 I/O 不会比 字节流 I/O操作复杂。输入和输出流的类与本地字符集进行自动转换。使用字符的程序来代替字节流可以自动适应本地字符集，并可以准备国际化，而这完全不需要程序员额外的工作。

如果国际化不是一个优先事项，你可以简单地使用字符流类，而不必太注意字符集问题。以后，如果国际化成为当务之急，你的程序可以方便适应这种需求的扩展。见国际化获取更多信息。

用法

字符流类描述在 Reader 和 Writer。而对应文件 I/O ，在 FileReader 和 FileWriter，下面是一个 CopyCharacters 例子：

```
public class CopyCharacters {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("resources/xanadu.txt");
            outputStream = new FileWriter("resources/characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters 与 CopyBytes 是非常相似的。最重要的区别在于 CopyCharacters 使用的 FileReader 和 FileWriter 用于输入输出，而 CopyBytes 使用 FileInputStream 和 FileOutputStream 中的。请注意，这两个CopyBytes和CopyCharacters使用int变量来读取和写入；在 CopyCharacters，int 变量保存在其最后的16位字符值;在 CopyBytes，int 变量保存在其最后的8位字节的值。

字符流使用字节流

字符流往往是对字节流的“包装”。字符流使用字节流来执行物理I/O，同时字符流处理字符和字节之间的转换。例如，FileReader 使用 FileInputStream，而 FileWriter使用的是 FileOutputStream。

有两种通用的字节到字符的“桥梁”流：InputStreamReader 和 OutputStreamWriter。当没有预包装的字符流类时，使用它们来创建字符流。在 socket 章节中将展示该用法。

面向行的 I/O

字符 I/O 通常发生在较大的单位不是单个字符。一个常用的单位是行：用行结束符结尾。行结束符可以是回车/换行序列（“ `\r\n` ”），一个回车（“ `\r` ”），或一个换行符（“ `\n` ”）。支持所有可能的行结束符，程序可以读取任何广泛使用的操作系统创建的文本文件。

修改 CopyCharacters 来演示如使用面向行的 I/O。要做到这一点，我们必须使用两个类，BufferedReader 和 PrintWriter 的。我们会在缓冲 I/O 和Formatting 章节更加深入地研究这些类。

该 CopyLines 示例调用 BufferedReader.readLine 和 PrintWriter.println 同时做一行的输入和输出。

```
public class CopyLines {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("resources/xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("resources/characteroutput.t
xt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

调用 readLine 按行返回文本行。CopyLines 使用 println 输出带有当前操作系统的行终止符的每一行。这可能与输入文件中不是使用相同的行终止符。

除字符和行之外，有许多方法来构造文本的输入和输出。欲了解更多信息，请参阅 Scanning 和 Formatting。

缓冲流（Buffered Streams）

缓冲流通过减少调用本地 API 的次数来优化的输入和输出。

目前为止，大多数时候我们到看到使用非缓冲 I/O 的例子。这意味着每次读或写请求是由基础 OS 直接处理。这可以使一个程序效率低得多，因为每个这样的请求通常引发磁盘访问，网络活动，或一些其它的操作，而这些是相对昂贵的。

为了减少这种开销，所以 Java 平台实现缓冲 I/O 流。缓冲输入流从被称为缓冲区（buffer）的存储器区域读出数据;仅当缓冲区是空时，本地输入 API 才被调用。同样，缓冲输出流，将数据写入到缓存区，只有当缓冲区已满才调用本机输出 API。

程序可以转换的非缓冲流为缓冲流，这里用非缓冲流对象传递给缓冲流类的构造器。

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

用于包装非缓存流的缓冲流类有4个：BufferedInputStream 和 BufferedOutputStream 用于创建字节缓冲字节流, BufferedReader 和 BufferedWriter 用于创建字符缓冲字节流。

刷新缓冲流

刷新缓冲区是指在某个缓冲的关键点就可以将缓冲输出，而不必等待它填满。

一些缓冲输出类通过一个可选的构造函数参数支持 autoflush（自动刷新）。当自动刷新开启，某些关键事件会导致缓冲区被刷新。例如，自动刷新 PrintWriter 对象在每次调用 println 或者 format 时刷新缓冲区。查看 Formatting 了解更多关于这些的方法。

如果要手动刷新流，请调用其 flush 方法。flush 方法可以用于任何输出流，但对非缓冲流是没有效果的。

扫描（Scanning）和格式化（Formatting）

扫描和格式化允许程序读取和写入格式化的文本。

I/O 编程通常涉及对人类喜欢的整齐的格式化数据进行转换。为了帮助您与这些琐事，Java 平台提供了两个API。scanning API 使用分隔符模式将其输入分解为标记。formatting API 将数据重新组合成格式良好的，人类可读的形式。

扫描

将其输入分解为标记

默认情况下，Scanner 使用空格字符分隔标记。（空格字符包括空格，制表符和行终止符。为完整列表，请参阅 Character.isWhitespace）。示例 ScanXan 读取 xanadu.txt 的单个词语并打印他们：

```
public class ScanXan {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("resources/xanadu.txt"))
);

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

虽然 Scanner 不是流，但你仍然需要关闭它，以表明你与它的底层流执行完成。

调用 useDelimiter() ,指定一个正则表达式可以使用不同的标记分隔符。例如,假设您想要标记分隔符是一个逗号，后面可以跟空格。你会调用

```
s.useDelimiter(",\\s*");
```

转换成独立标记

该 ScanXan 示例是将所有的输入标记为简单的字符串值。Scanner 还支持所有的 Java 语言的基本类型（除 char），以及 BigInteger 和 BigDecimal 的。此外，数字值可以使用千位分隔符。因此，在一个美国的区域设置，Scanner 能正确地读出字符串“32,767”作为一个整数值。

这里要注意的是语言环境，因为千位分隔符和小数点符号是特定于语言环境。所以，下面的例子将无法正常在所有的语言环境中，如果我们没有指定 scanner 应该用在美国地区工作。可能你平时并不用关心，因为你输入的数据通常来自使用相同的语言环境。可以使用下面的语句来设置语言环境：

```
s.useLocale(Locale.US);
```

该 ScanSum 示例是将读取的 double 值列表进行相加：

```
public class ScanSum {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        Scanner s = null;
        double sum = 0;

        try {
            s = new Scanner(new BufferedReader(new FileReader("resources/usnumbers.txt"
)));
            s.useLocale(Locale.US);

            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

输出为:1032778.74159

格式化

实现格式化流对象要么是 字符流类的 PrintWriter 的实例，或为字节流类的 PrintStream 的实例。

注：对于 *PrintStream* 对象，你很可能只需要 *System.out* 和 *System.err*。（请参阅命令行 *I/O*）当你需要创建一个格式化的输出流，请实例化 *PrintWriter*，而不是 *PrintStream*。

像所有的字节和字符流对象一样，PrintStream 和 PrintWriter 的实例实现了一套标准的 write 方法用于简单的字节和字符输出。此外，PrintStream 和 PrintWriter 的执行同一套方法，将内部数据转换成格式化输出。提供了两个级别的格式：

- print 和 println 在一个标准的方式里面格式化独立的值 。
- format 用于格式化几乎任何数量的格式字符串值，且具有多种精确选择。

print 和 println 方法

调用 print 或 println 输出使用适当 toString 方法变换后的值的单一值。我们可以看到这 Root 例子：

```
public class Root {
    /**
     * @param args
     */
    public static void main(String[] args) {
            int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

输出为：

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

在 i 和 r 变量格式化了两次：第一次在重载的 print 使用代码，第二次是由Java编译器转换码自动生成，它也利用了 toString。您可以用这种方式格式化任意值，但对于结果没有太多的控制权。

format 方法

该 format 方法用于格式化基于 format string（格式字符串） 多参。格式字符串包含嵌入了 format specifiers （格式说明)的静态文本;除非使用了格式说明，否则格式字符串输出不变。

格式字符串支持许多功能。在本教程中，我们只介绍一些基础知识。有关完整说明，请参阅 API 规范关于格式字符串语法。

Root2 示例在一个 format 调用里面设置两个值：

```
public class Root2 {
    /**
     * @param args
     */
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

输出为：The square root of 2 is 1.414214.

像本例中所使用的格式为：

- d 格式化整数值为小数
- f 格式化浮点值作为小数
- n 输出特定于平台的行终止符。

这里有一些其他的转换格式：

- x 格式化整数为十六进制值
- s 格式化任何值作为字符串
- tB 格式化整数作为一个语言环境特定的月份名称。

还有许多其他的转换。

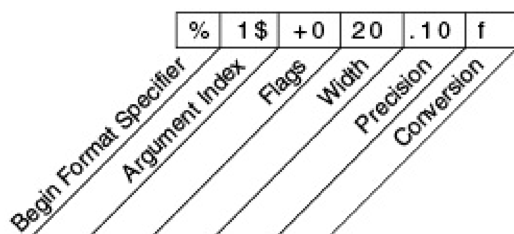
注意：除了 `%%` 和 `%n` ，其他格式符都要匹配参数，否则抛出异常。在 *Java* 编程语言中， `\ n` 转义总是产生换行符（ `\u000A` ）。不要使用 `\n` 除非你特别想要一个换行符。为了针对本地平台得到正确的行分隔符，请使用 `%n` 。

除了用于转换，格式说明符可以包含若干附加的元素，进一步定制格式化输出。下面是一个 Format 例子，使用一切可能的一种元素。

```
public class Format {
    /**
     * @param args
     */
    public static void main(String[] args) {
         System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

输出为：3.141593, +00000003.1415926536

附加元素都是可选的。下图显示了长格式符是如何分解成元素



元件必须出现在显示的顺序。从合适的工作，可选的元素是：

- Precision(精确)。对于浮点值，这是格式化值的数学精度。对于 s 和其他一般的转换，这是格式化值的最大宽度;该值右截断，如果有必要的。
- Width(宽度)。格式化值的最小宽度;如有必要，该值被填充。默认值是左用空格填充。
- Flags(标志)指定附加格式设置选项。在 Format 示例中，+ 标志指定的数量应始终标志格式，以及0标志指定0是填充字符。其他的标志包括 - （垫右侧）和（与区域特定的千位分隔符格式号）。请注意，某些标志不能与某些其他标志或与某些转换使用。
- Argument Index(参数索引)允许您指定的参数明确匹配。您还可以指定<到相同的参数作为前面的说明一致。这样的例子可以说：`System.out.format（“%F，%<+ 020.10f%N”，Math.PI）;`

命令行 I/O

命令行 I/O 描述了标准流（Standard Streams）和控制台（Console）对象。

Java 支持两种交互方式：标准流（Standard Streams）和通过控制台（Console）。

标准流

标准流是许多操作系统的一项功能。默认情况下，他们从键盘读取输入和写出到显示器。它们还支持对文件和程序之间的 I/O，但该功能是通过命令行解释器，而不是由程序控制。

Java平台支持三种标准流：标准输入（Standard Input, 通过 System.in 访问）、标准输出（Standard Output, 通过System.out 的访问）和标准错误（ Standard Error, 通过System.err 的访问）。这些对象被自动定义，并不需要被打开。标准输出和标准错误都用于输出;错误输出允许用户转移经常性的输出到一个文件中，仍然能够读取错误消息。

您可能希望标准流是字符流，但是，由于历史的原因，他们是字节流。 System.out 和 System.err 定义为 PrintStream 的对象。虽然这在技术上是一个字节流，PrintStream 利用内部字符流对象来模拟多种字符流的功能。

相比之下，System.in 是一个没有字符流功能的字节流。若要想将标准的输入作为字符流，可以包装 System.in 在 InputStreamReader

```
InputStreamReader cin = new InputStreamReader(System.in);
```

Console (控制台)

更先进的替代标准流的是 Console 。这个单一，预定义的 Console 类型的对象，有大部分的标准流提供的功能，另外还有其他功能。Console 对于安全的密码输入特别有用。Console 对象还提供了真正的输入输出字符流，是通过 reader 和 writer 方法实现的。

若程序想使用 Console ，它必须尝试通过调用 System.console() 检索 Console 对象。如果 Console 对象存在，通过此方法将其返回。如果返回 NULL，则 Console 操作是不允许的，要么是因为操作系统不支持他们或者是因为程序本身是在非交互环境中启动的。

Console 对象支持通过读取密码的方法安全输入密码。该方法有助于在两个方面的安全。第一，它抑制回应，因此密码在用户的屏幕是不可见的。第二，readPassword 返回一个字符数组，而不是字符串，所以，密码可以被覆盖，只要它是不再需要就可以从存储器中删除。

Password 例子是一个展示了更改用户的密码原型程序。它演示了几种 Console 方法

```
public class Password {
    /**
     * @param args
     */
    public static void main(String[] args) {
        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");

        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 = c.readPassword("Enter your new password: ");
                char [] newPassword2 = c.readPassword("Enter new password again: ");
                noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
                    change(login, newPassword1);
                    c.format("Password for %s changed.%n", login);
                }
                Arrays.fill(newPassword1, ' ');
                Arrays.fill(newPassword2, ' ');
            } while (noMatch);
        }

        Arrays.fill(oldPassword, ' ');
    }

    // Dummy change method.
    static boolean verify(String login, char[] password) {
        // This method always returns
        // true in this example.
        // Modify this method to verify
        // password according to your rules.
        return true;
    }

    // Dummy change method.
    static void change(String login, char[] password) {
        // Modify this method to change
        // password according to your rules.
    }
}
```

上面的流程是：

- 尝试检索 Console 对象。如果对象是不可用，中止。
- 调用 Console.readLine 提示并读取用户的登录名。
- 调用 Console.readPassword 提示并读取用户的现有密码。
- 调用 verify 确认该用户被授权可以改变密码。（在本例中，假设 verify 是总是返回true ）
- 重复下列步骤，直到用户输入的密码相同两次：
  - 调用 Console.readPassword 两次提示和读一个新的密码。
  - 如果用户输入的密码两次，调用 change 去改变它。 （同样，change 是一个虚拟的方法）
  - 用空格覆盖这两个密码。
- 用空格覆盖旧的密码。

数据流（Data Streams）

Data Streams 处理原始数据类型和字符串值的二进制 I/O。

支持基本数据类型的值（(boolean, char, byte, short, int, long, float, 和 double）以及字符串值的二进制 I/O。所有数据流实现 DataInput 或 DataOutput 接口。本节重点介绍这些接口的广泛使用的实现，DataInputStream 和 DataOutputStream 类。

DataStreams 例子展示了数据流通过写出的一组数据记录到文件，然后再次从文件中读取这些记录。每个记录包括涉及在发票上的项目，如下表中三个值：

记录中顺序	数据类型	数据描述	输出方法	输入方法
1	double	Item price	DataOutputStream.writeDouble	DataInputStream.readD
2	int	Unit count	DataOutputStream.writeInt	DataInputStream.readIr
3	String	Item description	DataOutputStream.writeUTF	DataInputStream.readU

首先，定义了几个常量，数据文件的名称，以及数据。

```
static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};
```

DataStreams 打开一个输出流，提供一个缓冲的文件输出字节流：

```
out = new DataOutputStream(new BufferedOutputStream(
              new FileOutputStream(dataFile)))
```

DataStreams 写出记录并关闭输出流：

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

该 writeUTF 方法写出以 UTF-8 改进形式的字符串值。

现在，DataStreams 读回数据。首先，它必须提供一个输入流，和变量来保存的输入数据。像 DataOutputStream 、DataInputStream 类，必须构造成一个字节流的包装器。

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

double price;
int unit;
String desc;
double total = 0.0;
```

现在，DataStreams 可以读取流里面的每个记录，并在遇到它时将数据报告出来：

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d" + " units of %s at $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

请注意，DataStreams 通过捕获 EOFException 检测文件结束的条件而不是测试无效的返回值。所有实现了 DataInput 的方法都使用 EOFException 类来代替返回值。

还要注意的是 DataStreams 中的各个 write 需要匹配对应相应的 read。它需要由程序员来保证。

DataStreams 使用了一个非常糟糕的编程技术：它使用浮点数来表示的货币价值。在一般情况下，浮点数是不好的精确数值。这对小数尤其糟糕，因为共同值（如 0.1），没有一个二进制的表示。

正确的类型用于货币值是 java.math.BigDecimal 的。不幸的是，BigDecimal 是一个对象的类型，因此它不能与数据流工作。然而，BigDecimal 将与对象流工作，而这部分内容将在下一节讲解。

对象流（Object Streams）

对象流处理对象的二进制 I/O。

正如数据流支持的是基本数据类型的 I/O，对象流支持的对象 I/O。大多数，但不是全部，标准类支持他们的对象的序列化，都需要实现 Serializable 接口。

对象流类包括 ObjectInputStream 和 ObjectOutputStream 的。这些类实现的 ObjectInput 与 ObjectOutput 的，这些都是 DataInput 和DataOutput 的子接口。这意味着，所有包含在数据流中的基本数据类型 I/O 方法也在对象流中实现了。这样一个对象流可以包含基本数据类型值和对象值的混合。该ObjectStreams 例子说明了这一点。ObjectStreams 创建与 DataStreams 相同的应用程序。首先，价格现在是 BigDecimal 对象，以更好地代表分数值。其次，Calendar 对象被写入到数据文件中，指示发票日期。

```
public class ObjectStreams {
    static final String dataFile = "invoicedata";

    static final BigDecimal[] prices = {
        new BigDecimal("19.99"),
```

```
            new BigDecimal("9.99"),
            new BigDecimal("15.99"),
            new BigDecimal("3.99"),
            new BigDecimal("4.99") };
    static final int[] units = { 12, 8, 13, 29, 50 };
    static final String[] descs = { "Java T-shirt",
            "Java Mug",
            "Duke Juggling Dolls",
            "Java Pin",
            "Java Key Chain" };

    public static void main(String[] args)
        throws IOException, ClassNotFoundException {


        ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(new
                    BufferedOutputStream(new FileOutputStream(dataFile)));

            out.writeObject(Calendar.getInstance());
            for (int i = 0; i < prices.length; i ++) {
                out.writeObject(prices[i]);
                out.writeInt(units[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }

        ObjectInputStream in = null;
        try {
            in = new ObjectInputStream(new
                    BufferedInputStream(new FileInputStream(dataFile)));

            Calendar date = null;
            BigDecimal price;
            int unit;
            String desc;
            BigDecimal total = new BigDecimal(0);

            date = (Calendar) in.readObject();

            System.out.format ("On %tA, %<tB %<te, %<tY:%n", date);

            try {
                while (true) {
                    price = (BigDecimal) in.readObject();
                    unit = in.readInt();
                    desc = in.readUTF();
                    System.out.format("You ordered %d units of %s at $%.2f%n",
                            unit, desc, price);
                    total = total.add(price.multiply(new BigDecimal(unit)));
```

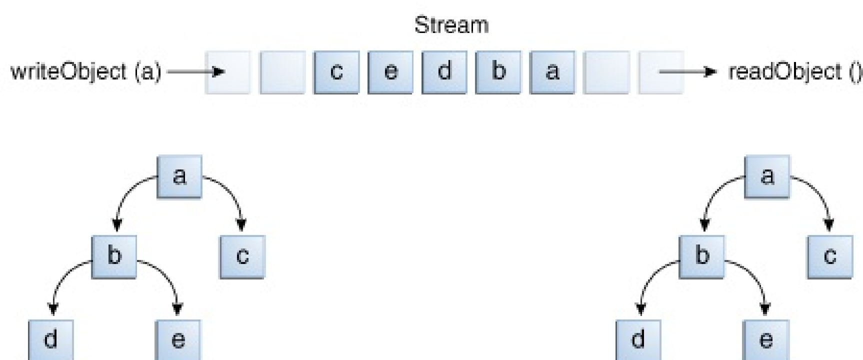
```
                }
            } catch (EOFException e) {}
            System.out.format("For a TOTAL of: $%.2f%n", total);
        } finally {
            in.close();
        }
    }
}
```

如果的 readObject() 不返回预期的对象类型，试图将它转换为正确的类型可能会抛出一个 ClassNotFoundException。在这个简单的例子，这是不可能发生的，所以我们不要试图捕获异常。相反，我们通知编译器，我们已经意识到这个问题，添加 ClassNotFoundException 到主方法的 throws 子句中的。

复杂对象的 I/O

writeObject 和 readObject 方法简单易用，但它们包含了一些非常复杂的对象管理逻辑。这不像 Calendar 类，它只是封装了原始值。但许多对象包含其他对象的引用。如果 readObject 从流重构一个对象，它必须能够重建所有的原始对象所引用的对象。这些额外的对象可能有他们自己的引用，依此类推。在这种情况下，writeObject 遍历对象引用的整个网络，并将该网络中的所有对象写入流。因此，writeObject 单个调用可以导致大量的对象被写入流。

如下图所示，其中 writeObject 调用名为 a 的单个对象。这个对象包含对象的引用 b和 c，而 b 包含引用 d 和 e。调用 writeObject(a) 写入的不只是一个 a，还包括所有需要重新构成的这个网络中的其他4个对象。当通过 readObject 读回 a 时，其他四个对象也被读回，同时，所有的原始对象的引用被保留。



如果在同一个流的两个对象引用了同一个对象会发生什么？流只包含一个对象的一个拷贝，尽管它可以包含任何数量的对它的引用。因此，如果你明确地写一个对象到流两次，实际上只是写入了2此引用。例如，如果下面的代码写入一个对象 ob 两次到流：

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

每个 writeObject 都对应一个 readObject， 所以从流里面读回的代码如下：

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

ob1 和 ob2 都是相同对象的引用。

然而，如果一个单独的对象被写入到两个不同的数据流，它被有效地复用 - 一个程序从两个流读回的将是两个不同的对象。

源码

本章例子的源码，可以在 https://github.com/waylau/essential-java 中 `com.waylau.essentialjava.io` 包下找到。

文件 I/O

本教程讲述的是 JDK 7 版本以来引入的新的 I/O 机制（也被称为 NIO.2）。

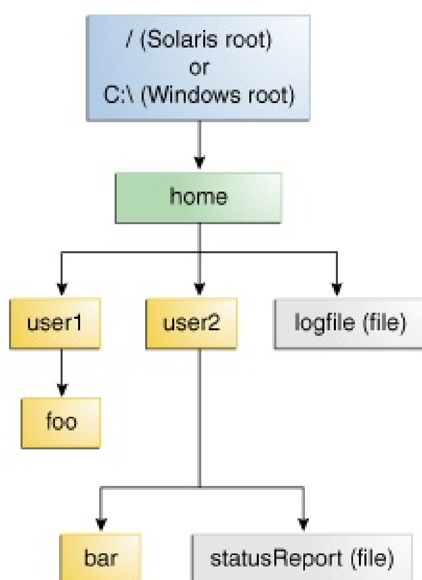
相关的包在 java.nio.file ，其中 java.nio.file.attribute 提供对文件 I/O 以及访问默认文件系统的全面支持。虽然 API 有很多类，但你只需要重点关注几个。你会看到，这个 API 是非常直观和易于使用。

什么是路径（Path）？在其他文件系统的实际是怎么样的？

文件系统是用某种媒体形式存储和组织文件，一般是一个或多个硬盘驱动器，以这样的方式，它们可以很容易地检索文件。目前使用的大多数文件系统存储文件是以树（或层次）结构。在树的顶部是一个（或多个）根节点。根节点下，有文件和目录（在Microsoft Windows 系统是指文件夹）。每个目录可以包含文件和子目录，而这又可以包含文件和子目录，以此类推，有可能是无限深度。

什么是路径（Path）？

下图显示了一个包含一个根节点的目录树。Microsoft Windows 支持多个根节点。每个根节点映射到一个卷，如 `C:\` 或 `D:\` 。 Solaris OS 支持一个根节点，这由斜杠 `/` 表示。



文件系统通过路径来确定文件。例如，上图 statusReport 在 Solaris OS 描述为：

```
/home/sally/statusReport
```

而在 Microsoft Windows 下，描述如下：

```
C:\home\sally\statusReport
```

用来分隔目录名称的字符（也称为分隔符）是特定于文件系统的：Solaris OS 中使用正斜杠（/），而 Microsoft Windows 使用反斜杠（\）。

相对或绝对路径？

路径可以是相对或绝对的。绝对路径总是包含根元素以及找到该文件所需要的完整的目录列表。对于例如，`/home/sally/statusReport` 是一个绝对路径。所有找到的文件所需的信息都包含在路径字符串里。

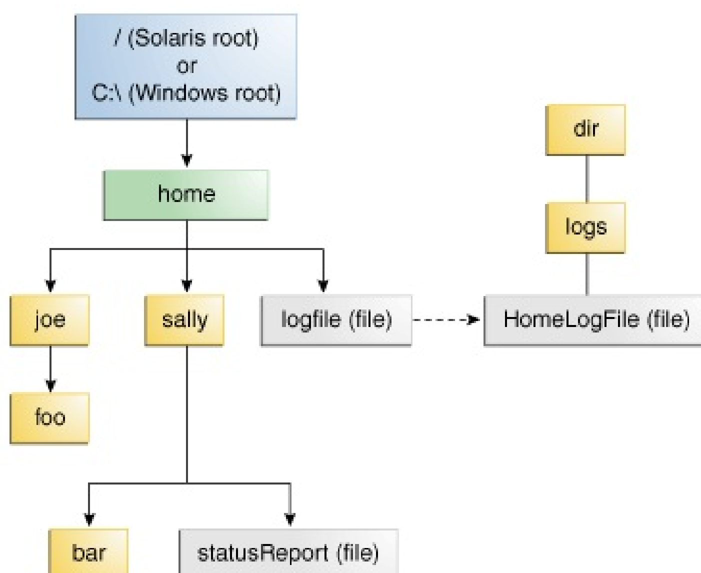
相对路径需要与另一路径进行组合才能访问到文件。例如，`joe/foo` 是一个相对路径,没有更多的信息，程序不能可靠地定位 `joe/foo` 目录。

符号链接（Symbolic Links）

文件系统对象最典型的是目录或文件。每个人都熟悉这些对象。但是，某些文件系统还支持符号链接的概念。符号链接也被称为符号链接（symlink）或软链接（soft link）。

符号链接是，作为一个引用到另一个文件的特殊文件。在大多数情况下，符号链接对于应用程序来说透明的，符号链接上面的操作会被自动重定向到链接的目标（链接的目标是指该所指向的文件或目录）。当符号链接删除或重命名，在这种情况下，链接本身被删除或重命名，而不是链接的目标。

在下图中，logFile 对于用户来说看起来似乎是一个普通文件，但它实际上是 `dir/logs/HomeLogFile` 文件的符号链接。HomeLogFile 是链接的目标。



符号链接通常对用户来说是透明。读取或写入符号链接是和读取或写入到任何其他文件或目录是一样的。

解析链接（resolving a link）是指在文件系统中用实际位置取代符号链接。在这个例子中，logFile 解析为 `dir/logs/HomeLogFile`

在实际情况下，大多数文件系统自由使用的符号链接。有时，一不小心创建符号链接会导致循环引用。循环引用是指，当链接的目标点回到原来的链接。循环引用可能是间接的：目录 a 指向目录 b，b 指向目录 c，其中包含的子目录指回目录 a 。当一个程序被递归遍历目录结构时，循环引用可能会导致混乱。但是，这种情况已经做了限制，不会导致程序无限循环。

接下来章节将讨论 Java 文件 I/O 的核心 Path 类。

Path 类

该 Path 类是从 Java SE 7 开始引入的， 是 java.nio.file 包的主要进入点之一。

注：若果 *Java SE 7* 之前的版本，可以使用 *File.toPath* 实现 *Path* 类似的功能

Path 类是在文件系统路径的编程表示。Path 对象包含了文件名和目录列表，用于构建路径，以及检查，定位和操作文件。

Path 实例反映了基础平台。在 Solaris OS,路径使用 Solaris 语法（ `/home/joe/foo` ）,而在 Microsoft Windows，路径使用 Windows 语法（ `C:\home\joe\foo` ）。路径是与系统相关，即 Solaris 文件系统中的路径不能与 Windows 文件系统的路径进行匹配。

对应于该路径的文件或目录可能不存在。您可以创建一个 Path 实例，并以各种方式操纵它：您可以附加到它，提取它的一部分，把它比作其他路径。在适当的时候，可以使用在 Files 类的方法来检查对应路径的文件是否存在，创建文件，打开它，删除它，改变它的权限，等等。

Path 操作

Path 类包括各种方法，可用于获得关于路径信息，路径的接入元件，路径转换为其它形式，或提取路径的部分。也有用于匹配的路径字符串的方法，也有用于在一个路径去除冗余的方法。这些路径方法，有时也被称为语义操作（syntactic operations），因为是在它们的路径本身进行操作，而不是访问文件系统。

创建路径

Path 实例包含用于指定文件或目录的位置的信息。在它被定义的时候，一个 Path 上设置了一系列的一个或多个名称。根元素或文件名可能被包括在内，但也不是必需的。Path 可能包含只是一个单一的目录或文件名。

您可以通过 Paths（注意是复数）助手类的 get 方法很容易地创建一个 Path 对象：

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

Paths.get 是下面方式的简写：

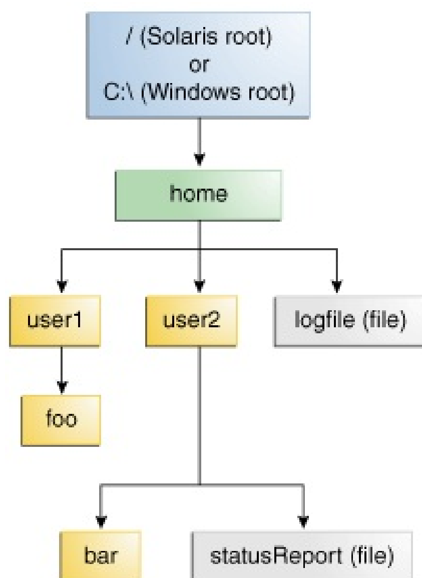
```
Path p4 = FileSystems.getDefault().getPath("/users/sally");
```

下面的例子假设你的 home 目录是 `/u/joe` ,则将创建 `/u/joe/logs/foo.log` , 或若果是 Windows 环境，则为 `C:\joe\logs\foo.log`

检索有关一个路径

你可以把路径作为储存这些名称元素的序列。在目录结构中的最高元素将设在索引为0的目录结构中，而最低元件将设在索引 [n-1]，其中 n 是 Path 的元素个数。方法可用于检索各个元素或使用这些索引 Path 的子序列。

本示例使用下面的目录结构：



下面的代码片段定义了一个 Path 实例，然后调用一些方法来获取有关的路径信息：

```
// None of these methods requires that the file corresponding
// to the Path exists.
// Microsoft Windows syntax
Path path = Paths.get("C:\\home\\joe\\foo");

// Solaris syntax
Path path = Paths.get("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

下面是 Windows 和 Solaris OS 不同的输出:

方法	**Solaris OS 返回**	**Microsoft Windows 返回**
toString	/home/joe/foo	C:\home\joe\foo
getFileName	foo	foo
getName(0)	home	home
getNameCount	3	3
subpath(0,2)	home/joe	home\joe
getParent	/home/joe	\home\joe
getRoot	/	C:\

下面是一个相对路径的例子：

```
// Solaris syntax
Path path = Paths.get("sally/bar");
or
// Microsoft Windows syntax
Path path = Paths.get("sally\\bar");
```

下面是 Windows 和 Solaris OS 不同的输出:

方法	Solaris OS 返回	Microsoft Windows 返回
toString	sally/bar	sally\bar
getFileName	bar	bar
getName(0)	sally	sally
getNameCount	2	2
subpath(0,1)	sally	sally
getParent	sally	sally
getRoot	null	null

从 Path 中移除冗余

许多文件系统使用“.”符号表示当前目录，“..”来表示父目录。您可能有一个 Path 包含冗余目录信息的情况。也许一个服务器配置为保存日志文件在“/dir/logs/.”目录，你想删除后面的“/.”

下面的例子都包含冗余：

```
/home/./joe/foo
/home/sally/../joe/foo
```

normalize 方法是删除任何多余的元素，其中包括任何 出现的“.”或“directory/...”。前面的例子规范化为 `/home/joe/foo`

要注意，当它清理一个路径时，normalize 不检查文件系统。这是一个纯粹的句法操作。在第二个例子中，如果 sally 是一个符号链接，删除 `sally/..` 可能会导致不能定位的预期文件。

清理路径的同时，你可以使用 toRealPath 方法来确保结果定位正确的文件。此方法在下一节中描述

转换一个路径

可以使用3个方法来转换路径。toUri 将路径转换为可以在浏览器中打开一个字符串，例如：

```
Path p1 = Paths.get("/home/logfile");
// Result is file:///home/logfile
System.out.format("%s%n", p1.toUri());
```

toAbsolutePath 方法将路径转为绝对路径。如果传递的路径已是绝对的，则返回同一Path 对象。toAbsolutePath 方法可以非常有助于处理用户输入的文件名。例如

```
public class FileTest {
    /**
     * @param args
     */
    public static void main(String[] args) {

        if (args.length < 1) {
            System.out.println("usage: FileTest file");
            System.exit(-1);
        }

        // Converts the input string to a Path object.
        Path inputPath = Paths.get(args[0]);

        // Converts the input Path
        // to an absolute path.
        // Generally, this means prepending
        // the current working
        // directory.  If this example
        // were called like this:
        //     java FileTest foo
        // the getRoot and getParent methods
        // would return null
        // on the original "inputPath"
        // instance.  Invoking getRoot and
        // getParent on the "fullPath"
        // instance returns expected values.
        Path fullPath = inputPath.toAbsolutePath();
    }
}
```

该 toAbsolutePath 方法转换用户输入并返回一个 Path 对于查询时返回是有用的值。此方法不需要文件存在才能正常工作。

toRealPath 方法返回一个已经存在文件的真实的路径，此方法执行以下其中一个：

- 如果 true 被传递到该方法，同时文件系统支持符号链接，那么该方法可以解析路径中的任何符号链接。
- 如果 Path 是相对的，它返回一个绝对路径。
- 如果 Path 中包含任何冗余元素，则返回一个删除冗余元素的路径。

若文件不存在或者无法访问,则方法抛出异常。可以捕捉处理异常：

```
try {
    Path fp = path.toRealPath();
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
    // Logic for case when file doesn't exist.
} catch (IOException x) {
    System.err.format("%s%n", x);
    // Logic for other sort of file error.
}
```

连接两个路径

可以使用 resolve 连接两个路径。你传递一个局部路径（partial path, 不包括一个根元素的路径），可以将局部路径追加到原始的路径。

例如，请考虑下面的代码片段：

```
// Solaris
Path p1 = Paths.get("/home/joe/foo");
// Result is /home/joe/foo/bar
System.out.format("%s%n", p1.resolve("bar"));

or

// Microsoft Windows
Path p1 = Paths.get("C:\\home\\joe\\foo");
// Result is C:\home\joe\foo\bar
System.out.format("%s%n", p1.resolve("bar"));
```

传递相对路径到 resolve 方法返回路径中的传递路径：

```
// Result is /home/joe
Paths.get("foo").resolve("/home/joe");
```

在两个路径间创建路径

文件 I/O 代码中的常见的需求是从路径能在不同的文件系统中兼容。可以使用relativize 方法满足这一点。新的路径是相对于原来的路径。

例如，定义为 joe 和 sally 相对路径：

```
Path p1 = Paths.get("joe");
Path p2 = Paths.get("sally");
```

在没有任何其他信息的，假定 joe 和 sally 是同一级别的节点。从 joe 导航到 sally，你会希望首先导航上一级父节点，然后向下找到 sally：

```
// Result is ../sally
Path p1_to_p2 = p1.relativize(p2);
// Result is ../joe
Path p2_to_p1 = p2.relativize(p1);
```

下面是复杂点的例子:

```
Path p1 = Paths.get("home");
Path p3 = Paths.get("home/sally/bar");
// Result is sally/bar
Path p1_to_p3 = p1.relativize(p3);
// Result is ../..
Path p3_to_p1 = p3.relativize(p1);
```

Copy 是个完整的使用 relativize 和 resolve 的例子：

```
public class Copy {

    /**
     * Returns {@code true} if okay to overwrite a  file ("cp -i")
     */
    static boolean okayToOverwrite(Path file) {
        String answer = System.console().readLine("overwrite %s (yes/no)? ", file);
        return (answer.equalsIgnoreCase("y") || answer.equalsIgnoreCase("yes"));
    }

    /**
     * Copy source file to target location. If {@code prompt} is true then
     * prompt user to overwrite target if it exists. The {@code preserve}
     * parameter determines if file attributes should be copied/preserved.
     */
    static void copyFile(Path source, Path target, boolean prompt, boolean preserve) {
        CopyOption[] options = (preserve) ?
            new CopyOption[] { COPY_ATTRIBUTES, REPLACE_EXISTING } :
            new CopyOption[] { REPLACE_EXISTING };
        if (!prompt || Files.notExists(target) || okayToOverwrite(target)) {
            try {
                Files.copy(source, target, options);
            } catch (IOException x) {
                System.err.format("Unable to copy: %s: %s%n", source, x);
            }
        }
    }

    /**
     * A {@code FileVisitor} that copies a file-tree ("cp -r")
```

```
     */
    static class TreeCopier implements FileVisitor<Path> {
        private final Path source;
        private final Path target;
        private final boolean prompt;
        private final boolean preserve;

        TreeCopier(Path source, Path target, boolean prompt, boolean preserve) {
            this.source = source;
            this.target = target;
            this.prompt = prompt;
            this.preserve = preserve;
        }

        @Override
        public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
{
            // before visiting entries in a directory we copy the directory
            // (okay if directory already exists).
            CopyOption[] options = (preserve) ?
                new CopyOption[] { COPY_ATTRIBUTES } : new CopyOption[0];

            Path newdir = target.resolve(source.relativize(dir));
            try {
                Files.copy(dir, newdir, options);
            } catch (FileAlreadyExistsException x) {
                // ignore
            } catch (IOException x) {
                System.err.format("Unable to create: %s: %s%n", newdir, x);
                return SKIP_SUBTREE;
            }
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {
            copyFile(file, target.resolve(source.relativize(file)),
                     prompt, preserve);
            return CONTINUE;
        }

        @Override
        public FileVisitResult postVisitDirectory(Path dir, IOException exc) {
            // fix up modification time of directory when done
            if (exc == null && preserve) {
                Path newdir = target.resolve(source.relativize(dir));
                try {
                    FileTime time = Files.getLastModifiedTime(dir);
                    Files.setLastModifiedTime(newdir, time);
                } catch (IOException x) {
                    System.err.format("Unable to copy all attributes to: %s: %s%n", ne
wdir, x);
                }
```

```
            }
            return CONTINUE;
        }

        @Override
        public FileVisitResult visitFileFailed(Path file, IOException exc) {
            if (exc instanceof FileSystemLoopException) {
                System.err.println("cycle detected: " + file);
            } else {
                System.err.format("Unable to copy: %s: %s%n", file, exc);
            }
            return CONTINUE;
        }
    }

    static void usage() {
        System.err.println("java Copy [-ip] source... target");
        System.err.println("java Copy -r [-ip] source-dir... target");
        System.exit(-1);
    }

    public static void main(String[] args) throws IOException {
        boolean recursive = false;
        boolean prompt = false;
        boolean preserve = false;

        // process options
        int argi = 0;
        while (argi < args.length) {
            String arg = args[argi];
            if (!arg.startsWith("-"))
                break;
            if (arg.length() < 2)
                usage();
            for (int i=1; i<arg.length(); i++) {
                char c = arg.charAt(i);
                switch (c) {
                    case 'r' : recursive = true; break;
                    case 'i' : prompt = true; break;
                    case 'p' : preserve = true; break;
                    default : usage();
                }
            }
            argi++;
        }

        // remaining arguments are the source files(s) and the target location
        int remaining = args.length - argi;
        if (remaining < 2)
            usage();
        Path[] source = new Path[remaining-1];
        int i=0;
        while (remaining > 1) {
```

```
            source[i++] = Paths.get(args[argi++]);
            remaining--;
        }
        Path target = Paths.get(args[argi]);

        // check if target is a directory
        boolean isDir = Files.isDirectory(target);

        // copy each source file/directory to target
        for (i=0; i<source.length; i++) {
            Path dest = (isDir) ? target.resolve(source[i].getFileName()) : target;

            if (recursive) {
                // follow links when copying files
                EnumSet<FileVisitOption> opts = EnumSet.of(FileVisitOption.FOLLOW_LINK
S);
                TreeCopier tc = new TreeCopier(source[i], dest, prompt, preserve);
                Files.walkFileTree(source[i], opts, Integer.MAX_VALUE, tc);
            } else {
                // not recursive so source must not be a directory
                if (Files.isDirectory(source[i])) {
                    System.err.format("%s: is a directory%n", source[i]);
                    continue;
                }
                copyFile(source[i], dest, prompt, preserve);
            }
        }
    }
}
```

比较两个路径

Path 类支持 equals，从而使您能够测试两个路径是否相等。startsWith 和 endsWith 方法，可以测试路径中是否有特定的字符串开头或者结尾。这些方法很容易使用。 例如：

```
Path path = ...;
Path otherPath = ...;
Path beginning = Paths.get("/home");
Path ending = Paths.get("foo");

if (path.equals(otherPath)) {
    // equality logic here
} else if (path.startsWith(beginning)) {
    // path begins with "/home"
} else if (path.endsWith(ending)) {
    // path ends with "foo"
}
```

Path 类实现了 Iterable 接口。 iterator 方法返回一个对象，使您可以遍历路径中的元素名。返回的第一个元素是最接近目录树的根。下面的代码片段遍历路径，打印每个name 元素：

```
Path path = ...;
for (Path name: path) {
    System.out.println(name);
}
```

该类同时还是实现了 Comparable 接口，可以 compareTo 方法对于排序过的 Path 对象进行比较 。

你也可以把 Path 对象到放到 Collection。见集合线索有关此强大功能的更多信息。

如果您想验证两个 Path 对象是否定位同一个文件，可以使用 isSameFile 方法。

File(文件)操作

Files 类是 java.nio.file 包的其他主要入口。这个类提供了一组丰富的静态方法，用于读取、写入和操作文件和目录。Files 方法可以作用于 Path 对象实例。要进入下章节的学习，首先要建立如下概念：

释放系统资源

有许多使用此 API 的资源，如流或信道的，都实现或者继承了 java.io.Closeable 接口。一个 Closeable 的资源需在不用时调用 close 方法以释放资源。忘记关闭资源对应用程序的性能可能产生负面影响。

捕获异常

所有方法访问文件系统都可以抛出 IOException。最佳实践是通过 `try-with-resources` 语句（Java SE 7 引入该语句）来捕获异常。

使用 `try-with-resources` 语句的好处是，在资源不需要时，编译器会自动生成的代码以关闭资源。下面的代码显示了如何用：

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

或者，你可以使用 try-catch-finally 语句，在 finally 块记得关闭它们。例子如下：

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
BufferedWriter writer = null;
try {
    writer = Files.newBufferedWriter(file, charset);
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
} finally {
    if (writer != null) writer.close();
}
```

除了 IOException 异常，许多异常都继承了 FileSystemException。这个类有一些有用的方法，如返回所涉及的文件 (getFile)，详细信息字符串（getMessage），文件系统操作失败的原因（getReason），以及所涉及的“其他”的文件，如果有的话（getOtherFile）。

下面的代码片段显示了 getFile 方法的使用：

```
try (...) {
    ...
} catch (NoSuchFileException x) {
    System.err.format("%s does not exist\n", x.getFile());
}
```

可变参数

Files 方法可以接受 可变参数，用法如

```
Path Files.move(Path, Path, CopyOption...)
```

可变参数可以用逗号隔开的数组 (CopyOption[])，用法：

```
import static java.nio.file.StandardCopyOption.*;

Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

原子操作

几个 Files 的方法，如 move，是可以在某些文件系统上执行某些原子操作的。

原子文件操作是不能被中断或“部分”进行的操作。无论是执行或操作失败的整个操作。当你对文件系统的同一区域运行多个进程，并且需要保证每个进程访问一个完整的文件，这是非常重要的。

原子文件操作是不能被中断或不能进行“部分”的操作。整个操作要不就执行不要就操作失败。在多个进程中操作相同的文件系统，需要保证每个进程访问一个完整的文件，这是非常重要的。

方法链

许多文件 I/O 支持方法链。例如：

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService().
         lookupPrincipalByName("me");
```

该技术可以生成紧凑的代码，使您避免声明不需要临时变量。

什么是 **Glob**？

Glob 是一种模式，它使用通配符来指定文件或者目录名名称。例如：`*.java` 就是一个简单的 Glob，它指定了所有扩展名为“java”的文件。其中

- `*` 表示“任意的字符或字符组成字符串”
- `**` 原理类似于 `*`，但可以越过目录。此语法通常用于匹配的完整路径。
- `?` 表示“任意单个字符”
- 大括号指定子模式的集合。 例如：
  - {sun,moon,stars} 匹配 "sun", "moon", 或 "stars"
  - {temp,*tmp*} 匹配所有 "temp" 或 "tmp" 开头的字符串
- 方括号传达了单个字符集合，或者使用连字符（-）时的字符的范围。 例如：
  - [aeiou]匹配任意小写元音。
  - [0-9]匹配任意数字。
  - [A-Z]匹配任意大写字母。
  - [a-z,A-Z]匹配任何大写或小写字母。

在方括号中，`*`，`?`，和 `\` 与自身匹配。

- 所有其他字符与自身匹配。
- 要匹配 `*`，`?` 或其他特殊字符，您可以用反斜杠字符转义 `\` 。例如：`\\` 匹配一个反斜杠，`\?` 匹配问号。

下面是一些 Glob 的一些例子：

- `*.html` - 匹配结尾以 .html 的所有字符串
- `???` - 匹配所有的字符串恰好有三个字母或数字
- `*[0-9]*` - 匹配含有数字值的所有字符串
- `*.{htm,html,pdf}` - 匹配具有的 .htm 或 .html 或 .pdf 结尾的字符串
- `a?*.java?` - 匹配 a 开头随后跟至少一个字母或数字，并以 .java 结尾的字符串
- `{foo*,*[0-9]*}` - 匹配任何以 foo 开头的或包含数值的字符串

Glob 模式源于Unix操作系统，Unix 提供了一个“global命令”,它可以缩写为glob。Glob 模式与正则表达式类似，但它的功能有限。 详见 FileSystem 类的 getPathMatcher 。

链接意识

Files 方法在遇到符号链接时，要检测做什么，或者提供启用怎样的配置选项。

检查文件或目录

验证文件或者目录是否存在

使用 exists(Path, LinkOption...) 和 the notExists(Path, LinkOption...) 方法。注意 !Files.exists(path) 不同等于 Files.notExists(path)。当您在验证文件是否存在，三种可能的结果：

- 该文件被确认存在
- 该文件被证实不存在的
- 该文件的状态未知。当程序没有访问该文件时，可能会发生此结果。

若 exists 和 notExists 同时返回 false，则该文件的是否存在不能被验证。

检查是否可访问

使用 isReadable(Path), isWritable(Path), 和 isExecutable(Path) 来验证程序是否可以访问文件。

下面的代码片段验证一个特定的文件是否存在，以及该程序能够执行该文件：

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
         Files.isReadable(file) & Files.isExecutable(file);
```

注：一旦这些方法中的任何一个完成，就无法再保证文件是可以访问的了。所以，在许多应用程序中的一个共同安全缺陷是先执行一个检查，然后访问该文件。有关更多信息，使用搜索引擎查找 *TOCTTOU*

检查是否有两个路径定位了相同的文件

在使用符号链接的文件系统中，就可能有两个定位到相同文件的不同的路径。使用 isSameFile(Path, Path) 方法比较两个路径，以确定它们在该文件系统上是否定位为同一个文件。 例如：

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
}
```

删除文件或目录

您可以删除文件，目录或链接。如果是符号链接，则该链接被删除后，不会删除所链接的目标。对于目录来说，该目录必须是空的，否则删除失败。

Files 类提供了两个删除方法。

delete(Path) 方法删除文件或者删除失败将引发异常。例如，如果文件不存在就抛出 NoSuchFileException。您可以捕获该异常，以确定为什么删除失败，如下所示：

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

deleteIfExists(Path) 同样是删除文件，但在文件不存在时不会抛出异常。这在多个线程处理删除文件又不想抛出异常是很有用的。

复制文件或目录

使用 copy(Path, Path, CopyOption...) 方法。如果目标文件已经存在了，则复制就会失败，除非指定 REPLACE_EXISTING 选项来替换已经存在的文件。

目录可以被复制。但是，目录内的文件不会被复制，因此新目录是空的，即使原来的目录中包含的文件。

当复制一个符号链接，链接的目标被复制。如果你想复制链接本身而不是链接的内容，请指定的 NOFOLLOW_LINKS 或 REPLACE_EXISTING 选项。

这种方法需要一个可变参数的参数。下面 StandardCopyOption 和 LinkOption 枚举是支持的：

- REPLACE_EXISTING - 执行复制，即使目标文件已经存在。如果目标是一个符号链接，则链接本身被复制（而不是链接所指向的目标）。如果目标是一个非空目录，复制失败抛出 FileAlreadyExistsException。
- COPY_ATTRIBUTES - 复制文件属性复制到目标文件。所支持的准确的文件属性是和文件系统和平台相关的，但是 last-modified-time 是支持跨平台的，将被复制到目标文件。
- NOFOLLOW_LINKS - 指示符号链接不应该被跟随。如果要复制的文件是一个符号链接，该链接被复制（而不是链接的目标）

下面演示了 copy 的用法：

```
import static java.nio.file.StandardCopyOption.*;
...
Files.copy(source, target, REPLACE_EXISTING);
```

其他方法还包括，copy(InputStream, Path, CopyOptions...) 方法可用于所有字节从输入流复制到文件中。 copy(Path, OutputStream) 方法可用于所有字节从一个文件复制到输出流中。

移动一个文件或目录

使用 move(Path, Path, CopyOption...) 方法。如果目标文件已经存在，则移动失败，除非指定了 REPLACE_EXISTING 选项

空目录可以移动。如果该目录不为空，那么在移动时可以选择只移动该目录而不移动该目录中的内容。在 UNIX 系统中，移动在同一分区内的目录一般包括重命名的目录。在这种情况下，即使该目录包含文件，这方法仍然可行。

该方法采用可变参数的参数 - 以下 StandardCopyOption 枚举的支持：

- REPLACE_EXISTING - 执行移动，即使目标文件已经存在。如果目标是一个符号链接，符号链接被替换，但它指向的目标是不会受到影响。
- ATOMIC_MOVE - 此举为一个原子文件操作。如果文件系统不支持原子移动，将引发异常。在 ATOMIC_MOVE 选项下，将文件移动到一个目录时，可以保证任何进程访问目录

时都看到的是一个完整的文件。

下面介绍如何使用 move 方法：

```
import static java.nio.file.StandardCopyOption.*;
...
Files.move(source, target, REPLACE_EXISTING);
```

管理元数据（文件和文件存储的属性）

阅读，写作，和创建文件

随机访问文件

创建和读取目录

链接，符号或否则

走在文件树

查找文件

看目录的更改

其他有用的方法

传统的文件I/ O代码

并发

计算机用户想当然地认为他们的系统在一个时间可以做多件事。他们认为，他们可以工作在一个字处理器，而其他应用程序在下载文件，管理打印队列和音频流。即使是单一的应用程序通常也是被期望在一个时间来做多件事。例如，音频流应用程序必须同时读取数字音频，解压，管理播放，并更新显示。即使字处理器应该随时准备响应键盘和鼠标事件，不管多么繁忙，它总是能格式化文本或更新显示。可以做这样的事情的软件称为并发软件（concurrent software）。

在 Java 平台是完全支持并发编程。自从 5.0 版本以来，这个平台还包括高级并发 API, 主要集中在 java.util.concurrent 包。

源码

本章例子的源码，可以在 https://github.com/waylau/essential-java 中 com.waylau.essentialjava.concurrency 包下找到。

进程（Processes ）和线程（Threads）

进程和线程是并发编程的两个基本的执行单元。在 Java 中，并发编程主要涉及线程。

一个计算机系统通常有许多活动的进程和线程。在给定的时间内，每个处理器只能有一个线程得到真正的运行。对于单核处理器来说，处理时间是通过时间切片来在进程和线程之间进行共享的。

现在多核处理器或多进程的电脑系统越来越流行。这大大增强了系统的进程和线程的并发执行能力。但即便是没有多处理器或多进程的系统中，并发仍然是可能的。

进程

进程有一个独立的执行环境。进程通常有一个完整的、私人的基本运行时资源;特别是,每个进程都有其自己的内存空间。

进程往往被视为等同于程序或应用程序。然而,用户将看到一个单独的应用程序可能实际上是一组合作的进程。大多数操作系统都支持进程间通信(Inter Process Communication，简称 IPC)资源,如管道和套接字。IPC 不仅用于同个系统的进程之间的通信，也可以用在不同系统的进程。

大多数 Java 虚拟机的实现作为一个进程运行。Java 应用程序可以使用 ProcessBuilder 对象创建额外的进程。多进程应用程序超出了本书的讲解范围。

线程

线程有时被称为轻量级进程。进程和线程都提供一个执行环境,但创建一个新的线程比创建一个新的进程需要更少的资源。

线程中存在于进程中,每个进程都至少一个线程。线程共享进程的资源,包括内存和打开的文件。这使得工作变得高效，但也存在了一个潜在的问题——通信。

多线程执行是 Java 平台的一个重要特点。每个应用程序都至少有一个线程,或者几个,如果算上“系统”的线程（负责内存管理和信号处理）那就更多。但从程序员的角度来看,你启动只有一个线程，称为主线程。这个线程有能力创建额外的线程。

线程对象

每个线程都与 Thread 类的一个实例相关联。有两种使用线程对象来创建并发应用程序的基本策略：

- 为了直接控制线程的创建和管理，简单地初始化线程，应用程序每次需要启动一个异步任务。
- 通过传递给应用程序任务给一个 executor，从而从应用程序的其他部分抽象出线程管理。

定义和启动一个线程

Java 中有两种方式创建 Thread 的实例：

- 提供 Runnable 对象。Runnable 接口定义了一个方法 run ,用来包含线程要执行的代码。如 HelloRunnable 所示：

```
public class HelloRunnable implements Runnable {
    /* (non-Javadoc)
     * @see java.lang.Runnable#run()
     */
    @Override
    public void run() {
        System.out.println("Hello from a thread!");
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

- 继承 Thread。Thread 类本身是实现 Runnable，虽然它的 run 方法啥都没干。HelloThread 示例如下：

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }
    /**
     * @param args
     */
    public static void main(String[] args) {
        (new HelloThread()).start();
    }
}
```

请注意,这两个例子调用 start 来启动线程。

第一种方式,它使用 Runnable 对象,在实际应用中更普遍,因为 Runnable 对象可以继承 Thread 以外的类。第二种方式，在简单的应用程序更容易使用,但受限于你的任务类必须是一个 Thread 的后代。本书推荐使用第一种方法,将 Runnable 任务从 Thread 对象分离来执行任务。这不仅更灵活,而且它适用于高级线程管理 API。

Thread 类还定义了大量的方法用于线程管理。

Sleep 来暂停执行

Thread.sleep 可以让当前线程执行暂停一个时间段，这样处理器时间就可以给其他线程使用。

sleep 有两种重载形式：一个是指定睡眠时间为毫秒，另外一个是指定睡眠时间为纳秒级。然而，这些睡眠时间不能保证是精确的，因为它们是通过由操作系统来提供的，并受其限制，因而不能假设 sleep 的睡眠时间是精确的。此外，睡眠周期也可以通过中断终止，我们将在后面的章节中看到。

SleepMessages 示例使用 sleep 每隔4秒打印一次消息：

```
public class SleepMessages {

    /**
     * @param args
     */
    public static void main(String[] args) throws InterruptedException {
        String importantInfo[] = { "Mares eat oats", "Does eat oats", "Little lambs ea
t ivy",
                "A kid will eat ivy too" };

        for (int i = 0; i < importantInfo.length; i++) {

            // Pause for 4 seconds
            Thread.sleep(4000);

            // Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

请注意 main 声明抛出 InterruptedException。当 sleep 是激活的时候，若有另一个线程中断当前线程时，则 sleep 抛出异常。由于该应用程序还没有定义的另一个线程来引起的中断，所以考虑捕捉 InterruptedException。

中断（interrupt）

中断是表明一个线程，它应该停止它正在做和将要做的事。线程通过在 Thread 对象调用 interrupt 来实现线程的中断。为了中断机制能正常工作，被中断的线程必须支持自己的中断。

支持中断

如何实现线程支持自己的中断？这要看是它目前正在做什么。如果线程调用方法频繁抛出 InterruptedException 异常，那么它只要在 run 方法捕获了异常之后返回即可。例如 ：

```
for (int i = 0; i < importantInfo.length; i++) {

    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {

        // We've been interrupted: no more messages.
        return;
    }

    // Print a message
    System.out.println(importantInfo[i]);
}
```

很多方法都会抛出 InterruptedException，如 sleep，被设计成在收到中断时立即取消他们当前的操作并返回。

若线程长时间没有调用方法抛出 InterruptedException 的话，那么它必须定期调用 Thread.interrupted ，该方法在接收到中断后将返回 true。

```
for (int i = 0; i < inputs.length; i++) {

    heavyCrunch(inputs[i]);

    if (Thread.interrupted()) {

        // We've been interrupted: no more crunching.
        return;
    }
}
```

在这个简单的例子中，代码简单地测试该中断，如果已接收到中断线程就退出。在更复杂的应用程序，它可能会更有意义抛出一个 InterruptedException：

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

中断状态标志

中断机制是使用被称为中断状态的内部标志实现的。调用 Thread.interrupt 可以设置该标志。当一个线程通过调用静态方法 Thread.interrupted 来检查中断，中断状态被清除。非静态 isInterrupted 方法，它是用于线程来查询另一个线程的中断状态，而不会改变中断状态标志。

按照惯例，任何方法因抛出一个 InterruptedException 而退出都会清除中断状态。当然，它可能因为另一个线程调用 interrupt 而让那个中断状态立即被重新设置回来。

join 方法

join 方法允许一个线程等待另一个完成。假设 t 是一个正在执行的 Thread 对象，那么

```
t.join();
```

它会导致当前线程暂停执行直到 t 线程终止。join 允许程序员指定一个等待周期。与 sleep 一样，等待时间是依赖于操作系统的时间，同时不能假设 join 等待时间是精确的。

像 sleep 一样，join 并通过 InterruptedException 退出来响应中断。

SimpleThreads 示例

SimpleThreads 示例由两个线程。第一个线程是每个 Java 应用程序都有的主线程。主线程创建的 Runnable 对象 MessageLoop，并等待它完成。如果 MessageLoop 需要很长时间才能完成，主线程就中断它。

该 MessageLoop 线程打印出一系列消息。如果中断之前就已经打印了所有消息，则 MessageLoop 线程打印一条消息并退出。

```
public class SimpleThreads {

      // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
                          threadName,
                          message);
    }
```

```
    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too"
            };
            try {
                for (int i = 0; i < importantInfo.length; i++) {

                    // Pause for 4 seconds
                    Thread.sleep(4000);

                    // Print a message
                    threadMessage(importantInfo[i]);
                }
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }

    public static void main(String args[])
        throws InterruptedException {

        // Delay, in milliseconds before
        // we interrupt MessageLoop
        // thread (default one hour).
        long patience = 1000 * 60 * 60;

        // If command line argument
        // present, gives patience
        // in seconds.
        if (args.length > 0) {
            try {
                patience = Long.parseLong(args[0]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }
        }

        threadMessage("Starting MessageLoop thread");
        long startTime = System.currentTimeMillis();
        Thread t = new Thread(new MessageLoop());
        t.start();

        threadMessage("Waiting for MessageLoop thread to finish");

        // loop until MessageLoop
```

```
        // thread exits
        while (t.isAlive()) {
            threadMessage("Still waiting...");

            // Wait maximum of 1 second
            // for MessageLoop thread
            // to finish.

            t.join(1000);
            if (((System.currentTimeMillis() - startTime) > patience)
                  && t.isAlive()) {
                threadMessage("Tired of waiting!");
                t.interrupt();

                // Shouldn't be long now
                // -- wait indefinitely
                t.join();
            }
        }
        threadMessage("Finally!");
    }
}
```

同步（Synchronization）

线程间的通信主要是通过共享访问字段以及其字段所引用的对象来实现的。这种形式的通信是非常有效的，但可能导致2种可能的错误：线程干扰（thread interference）和内存一致性错误（memory consistency errors）。同步就是要需要避免这些错误的工具。

但是，同步可以引入线程竞争（thread contention），当两个或多个线程试图同时访问相同的资源时，并导致了 Java 运行时执行一个或多个线程更慢，或甚至暂停他们的执行。饥饿（Starvation）和活锁 （livelock） 是线程竞争的表现形式。

线程干扰

描述当多个线程访问共享数据时是错误如何出现。

考虑下面的一个简单的类 Counter：

```
public class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

其中的 increment 方法用来对 c 加1；decrement 方法用来对 c 减 1。然而，有多个线程中都存在对某个 Counter 对象的引用，那么线程间的干扰就可能导致出现我们不想要的结果。

线程间的干扰出现在多个线程对同一个数据进行多个操作的时候，也就是出现了“交错”。这就意味着操作是由多个步骤构成的，而此时，在这多个步骤的执行上出现了叠加。

Counter类对象的操作貌似不可能出现这种“交错(interleave)”，因为其中的两个关于c 的操作都很简单，只有一条语句。然而，即使是一条语句也是会被虚拟机翻译成多个步骤的。在这里，我们不深究虚拟机具体上上面的操作翻译成了什么样的步骤。只需要知道即使简单的 c++ 这样的表达式也是会被翻译成三个步骤的：

1. 获取 c 的当前值。

2. 对其当前值加 1。
3. 将增加后的值存储到 c 中。

表达式 c-- 也是会被按照同样的方式进行翻译，只不过第二步变成了减1，而不是加1。

假定线程 A 中调用 increment 方法，线程 B 中调用 decrement 方法，而调用时间基本上相同。如果 c 的初始值为 0，那么这两个操作的“交错”顺序可能如下：

1. 线程A：获取 c 的值。
2. 线程B：获取 c 的值。
3. 线程A：对获取到的值加1；其结果是1。
4. 线程B：对获取到的值减1；其结果是-1。
5. 线程A：将结果存储到 c 中；此时c的值是1。
6. 线程B：将结果存储到 c 中；此时c的值是-1。

这样线程 A 计算的值就丢失了，也就是被线程 B 的值覆盖了。上面的这种“交错”只是其中的一种可能性。在不同的系统环境中，有可能是 B 线程的结果丢失了，或者是根本就不会出现错误。由于这种“交错”是不可预测的，线程间相互干扰造成的 bug 是很难定位和修改的。

内存一致性错误

介绍了通过共享内存出现的不一致的错误。

内存一致性错误(Memory consistency errors)发生在不同线程对同一数据产生不同的“看法”。导致内存一致性错误的原因很复杂，超出了本书的描述范围。庆幸的是，程序员并不需要知道出现这些原因的细节。我们需要的是一种可以避免这种错误的方法。

避免出现内存一致性错误的关键在于理解 happens-before 关系。这种关系是一种简单的方法，能够确保一条语句对内存的写操作对于其它特定的语句都是可见的。为了理解这点，我们可以考虑如下的示例。假定定义了一个简单的 int 类型的字段并对其进行了初始化：

```
int counter = 0;
```

该字段由两个线程共享：A 和 B。假定线程 A 对 counter 进行了自增操作：

```
counter++;
```

然后，线程 B 打印 counter 的值：

```
System.out.println(counter);
```

如果以上两条语句是在同一个线程中执行的，那么输出的结果自然是1。但是如果这两条语句是在两个不同的线程中，那么输出的结构有可能是0。这是因为没有保证线程 A 对 counter 的修改对线程 B 来说是可见的。除非程序员在这两条语句间建立了一定的 happens-before 关系。

我们可以采取多种方式建立这种 happens-before 关系。使用同步就是其中之一，这点我们将会在下面的小节中看到。

到目前为止，我们已经看到了两种建立这种 happens-before 的方式：

- 当一条语句中调用了 Thread.start 方法，那么每一条和该语句已经建立了 happens-before 的语句都和新线程中的每一条语句有着这种 happens-before。引入并创建这个新线程的代码产生的结果对该新线程来说都是可见的。
- 当一个线程终止了并导致另外的线程中调用 Thread.join 的语句返回，那么此时这个终止了的线程中执行了的所有语句都与随后的 join 语句随后的所有语句建立了这种 happens-before 。也就是说终止了的线程中的代码效果对调用 join 方法的线程来说是可见。

关于哪些操作可以建立这种 happens-before，更多的信息请参阅“java.util.concurrent 包的概要说明”。

同步方法

描述了一个简单的做法，可以有效防止线程干扰和内存一致性错误。

Java 编程语言中提供了两种基本的同步用语：同步方法（synchronized methods）和同步语句（synchronized statements）。同步语句相对而言更为复杂一些，我们将在下一小节中进行描述。本节重点讨论同步方法。

我们只需要在声明方法的时候增加关键字 synchronized 即可：

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

如果 count 是 SynchronizedCounter 类的实例，设置其方法为同步方法将有两个效果：

- 首先，不可能出现对同一对象的同步方法的两个调用的“交错”。当一个线程在执行一个对象的同步方式的时候，其他所有的调用该对象的同步方法的线程都会被挂起，直到第一个线程对该对象操作完毕。
- 其次，当一个同步方法退出时，会自动与该对象的同步方法的后续调用建立 happens-before 关系。这就确保了对该对象的修改对其他线程是可见的。

注意：构造函数不能是 synchronized ——在构造函数前使用 synchronized 关键字将导致语义错误。同步构造函数是没有意义的。这是因为只有创建该对象的线程才能调用其构造函数。

警告：在创建多个线程共享的对象时，要特别小心对该对象的引用不能过早地“泄露”。例如，假定我们想要维护一个保存类的所有实例的列表 instances。我们可能会在构造函数中这样写到：

```
instances.add(this);
```

但是，其他线程可会在该对象的构造完成之前就访问该对象。

同步方法是一种简单的可以避免线程相互干扰和内存一致性错误的策略：如果一个对象对多个线程都是可见的，那么所有对该对象的变量的读写都应该是通过同步方法完成的（一个例外就是 final 字段，他在对象创建完成后是不能被修改的，因此，在对象创建完毕后，可以通过非同步的方法对其进行安全的读取）。这种策略是有效的，但是可能导致“活跃度（liveness）”问题。这点我们会在本课程的后面进行描述。

内部锁和同步

描述了一个更通用的同步方法，并介绍了同步是如何基于内部锁的。

同步是构建在被称为“内部锁（intrinsic lock）”或者是“监视锁（monitor lock）”的内部实体上的。（在 API 中通常被称为是“监视器（monitor）”。）内部锁在两个方面都扮演着重要的角色：保证对对象状态访问的排他性和建立也对象可见性相关的重要的“ happens-before。

每一个对象都有一个与之相关联动的内部锁。按照传统的做法，当一个线程需要对一个对象的字段进行排他性访问并保持访问的一致性时，他必须在访问前先获取该对象的内部锁，然后才能访问之，最后释放该内部锁。在线程获取对象的内部锁到释放对象的内部锁的这段时间，我们说该线程拥有该对象的内部锁。只要有一个线程已经拥有了一个内部锁，其他线程就不能再拥有该锁了。其他线程将会在试图获取该锁的时候被阻塞了。

当一个线程释放了一个内部锁，那么就会建立起该动作和后续获取该锁之间的 happens-before 关系。

同步方法中的锁

当一个线程调用一个同步方法的时候，他就自动地获得了该方法所属对象的内部锁，并在方法返回的时候释放该锁。即使是由于出现了没有被捕获的异常而导致方法返回，该锁也会被释放。

我们可能会感到疑惑：当调用一个静态的同步方法的时候会怎样了，静态方法是和类相关的，而不是和对象相关的。在这种情况下，线程获取的是该类的类对象的内部锁。这样对于静态字段的方法是通过一个和类的实例的锁相区分的另外的锁来进行的。

同步语句

另外一种创建同步代码的方式就是使用同步语句。和同步方法不同，使用同步语句是必须指明是要使用哪个对象的内部锁：

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

在上面的示例中，方法 addName 需要对 lastName 和 nameCount 的修改进行同步，还要避免同步调用其他对象的方法（在同步代码段中调用其他对象的方法可能导致“活跃度（Liveness）”中描述的问题）。如果没有使用同步语句，那么将不得不使用一个单独的，未同步的方法来完成对 nameList.add 的调用。

在改善并发性时，巧妙地使用同步语句能起到很大的帮助作用。例如，我们假定类 MsLunch 有两个实例字段，c1 和 c2，这两个变量绝不会一起使用。所有对这两个变量的更新都需要进行同步。但是没有理由阻止对 c1 的更新和对 c2 的更新出现交错——这样做会创建不必要的阻塞，进而降低并发性。此时，我们没有使用同步方法或者使用和this 相关的锁，而是创建了两个单独的对象来提供锁。

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

采用这种方式时需要特别的小心。我们必须绝对确保相关字段的访问交错是完全安全的。

重入同步（Reentrant Synchronization）

回忆前面提到的：线程不能获取已经被别的线程获取的锁。但是线程可以获取自身已经拥有的锁。允许一个线程能重复获得同一个锁就称为重入同步（reentrant synchronization）。它是这样的一种情况：在同步代码中直接或者间接地调用了还有同步代码的方法，两个同步代码段中使用的是同一个锁。如果没有重入同步，在编写同步代码时需要额外的小心，以避免线程将自己阻塞。

原子访问

介绍了不会被其他线程干扰的做法的总体思路。

在编程中，原子性动作就是指一次性有效完成的动作。原子性动作是不能在中间停止的：要么一次性完全执行完毕，要么就不执行。在动作没有执行完毕之前，是不会产生可见结果的。

通过前面的示例，我们已经发现了诸如 c++ 这样的自增表达式并不属于原子操作。即使是非常简单的表达式也包含了复杂的动作，这些动作可以被解释成许多别的动作。然而，的确存在一些原子操作的：

- 对几乎所有的原生数据类型变量（除了 long 和 double）的读写以及引用变量的读写都是原子的。
- 对所有声明为 volatile 的变量的读写都是原子的，包括 long 和 double 类型。

原子性动作是不会出现交错的，因此，使用这些原子性动作时不用考虑线程间的干扰。然而，这并不意味着可以移除对原子操作的同步。因为内存一致性错误还是有可能出现的。使用 volatile 变量可以减少内存一致性错误的风险，因为任何对 volatile 变 量的写操作都和后续对该变量的读操作建立了 happens-before 关系。这就意味着对 volatile 类型变量的修改对于别的线程来说是可见的。更重要的是，这意味着当一个线程读取一个 volatile 类型的变量时，他看到的不仅仅是对该变量的最后一次修改，还看到了导致这种修改的代码带来的其他影响。

使用简单的原子变量访问比通过同步代码来访问变量更高效，但是需要程序员的更多细心考虑，以避免内存一致性错误。这种额外的付出是否值得完全取决于应用程序的大小和复杂度。

活跃度（Liveness）

一个并行应用程序的及时执行能力被称为它的活跃度（liveness）。本节将介绍最常见的一种活跃度的问题——死锁，以及另外两个活跃度的问题——饥饿和活锁。

死锁（Deadlock）

死锁是指两个或两个以上的线程永远被阻塞,一直等待对方的资源。

下面是一个例子。

Alphonse 和 Gaston 是朋友,都很有礼貌。礼貌的一个严格的规则是,当你给一个朋友鞠躬时,你必须保持鞠躬,直到你的朋友鞠躬回给你。不幸的是,这条规则有个缺陷，那就是如果两个朋友同一时间向对方鞠躬，那就永远不会完了。这个示例应用程序中,死锁模型是这样的:

```
public class Deadlock {
    static class Friend {
        private final String name;

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s" + "  has bowed to me!%n", this.name, bower.getN
ame());
            bower.bowBack(this);
        }

        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s" + " has bowed back to me!%n", this.name, bower.
getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() {
                alphonse.bow(gaston);
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                gaston.bow(alphonse);
            }
        }).start();
    }
}
```

当他们尝试调用 bowBack 两个线程将被阻塞。无论是哪个线程永远不会结束，因为每个线程都在等待对方鞠躬。这就是死锁了。

饥饿和活锁（Starvation and Livelock）

饥饿和活锁虽比死锁问题稍微不常见点,但这些是在并发软件种每一个设计师仍然可能会遇到的问题。

饥饿（Starvation）

饥饿描述了这样一个情况,一个线程不能获得定期访问共享资源,于是无法继续执行。这种情况一般出现在共享资源被某些“贪婪”线程占用，而导致资源长时间不被其他线程可用。例如,假设一个对象提供一个同步的方法,往往需要很长时间返回。如果一个线程频繁调用该方法,其他线程若也需要频繁的同步访问同一个对象通常会被阻塞。

活锁（Livelock）

一个线程常常处于响应另一个线程的动作，如果其他线程也常常处于该线程的动作,那么就可能出现活锁。与死锁、活锁的线程一样，程序无法进一步执行。然而,线程是不会阻塞的，他们只是会忙于应对彼此的恢复工作。现实种的例子是，两人面对面试图通过一条走廊: Alphonse 移动到他的左则让路给 Gaston ,而 Gaston 移动到他的右侧想让 Alphonse 过去，两个人同时让路，但其实两人都挡住了对方没办法过去，他们仍然彼此阻塞。

Guarded Blocks

多线程之间经常需要协同工作，最常见的方式是使用 Guarded Blocks，它循环检查一个条件（通常初始值为 true），直到条件发生变化才跳出循环继续执行。在使用 Guarded Blocks 时有以下几个步骤需要注意：

假设 guardedJoy 方法必须要等待另一线程为共享变量 joy 设值才能继续执行。那么理论上可以用一个简单的条件循环来实现，但在等待过程中 guardedJoy 方法不停的检查循环条件实际上是一种资源浪费。

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

更加高效的保护方法是调用 Object.wait 将当前线程挂起，直到有另一线程发起事件通知（尽管通知的事件不一定是当前线程等待的事件）。

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

注意：一定要在循环里面调用 *wait* 方法，不要想当然的认为线程唤醒后循环条件一定发生了改变。

和其他可以暂停线程执行的方法一样，wait 方法会抛出 InterruptedException，在上面的例子中，因为我们关心的是 joy 的值，所以忽略了 InterruptedException。

为什么 guardedJoy 是 synchronized 的？假设 d 是用来调用 wait 的对象，当一个线程调用 d.wait，它必须要拥有 d的内部锁（否则会抛出异常），获得 d 的内部锁的最简单方法是在一个 synchronized 方法里面调用 wait。

当一个线程调用 wait 方法时，它释放锁并挂起。然后另一个线程请求并获得这个锁并调用 Object.notifyAll 通知所有等待该锁的线程。

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

当第二个线程释放这个该锁后，第一个线程再次请求该锁，从 wait 方法返回并继续执行。

注意：还有另外一个通知方法，notify()，它只会唤醒一个线程。但由于它并不允许指定哪一个线程被唤醒，所以一般只在大规模并发应用（即系统有大量相似任务的线程）中使用。因为对于大规模并发应用，我们其实并不关心哪一个线程被唤醒。

现在我们使用 Guarded blocks 创建一个生产者/消费者应用。这类应用需要在两个线程之间共享数据：生产者生产数据，消费者使用数据。两个线程通过共享对象通信。在这里，线程协同工作的关键是：生产者发布数据之前，消费者不能够去读取数据；消费者没有读取旧数据前，生产者不能发布新数据。

在下面的例子中，数据通过 Drop 对象共享的一系列文本消息：

```
public class Drop {
      // Message sent from producer
    // to consumer.
    private String message;
    // True if consumer should wait
    // for producer to send message,
    // false if producer should wait for
    // consumer to retrieve message.
    private boolean empty = true;

    public synchronized String take() {
        // Wait until message is
        // available.
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = true;
        // Notify producer that
        // status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message) {
        // Wait until message has
        // been retrieved.
        while (!empty) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        // Toggle status.
        empty = false;
        // Store message.
        this.message = message;
        // Notify consumer that status
        // has changed.
        notifyAll();
    }
}
```

Producer 是生产者线程，发送一组消息，字符串 DONE 表示所有消息都已经发送完成。为了模拟现实情况，生产者线程还会在消息发送时随机的暂停。

```
public class Producer implements Runnable {
    private Drop drop;

    public Producer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        String importantInfo[] = { "Mares eat oats", "Does eat oats", "Little lambs ea
t ivy",
                "A kid will eat ivy too" };
        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++) {
            drop.put(importantInfo[i]);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {
            }
        }
        drop.put("DONE");
    }
}
```

Consumer 是消费者线程，读取消息并打印出来，直到读取到字符串 DONE 为止。消费者线程在消息读取时也会随机的暂停。

```
public class Consumer implements Runnable {
    private Drop drop;

    public Consumer(Drop drop) {
        this.drop = drop;
    }

    public void run() {
        Random random = new Random();
        for (String message = drop.take(); !message.equals("DONE"); message = drop.tak
e()) {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try {
                Thread.sleep(random.nextInt(5000));
            } catch (InterruptedException e) {
            }
        }
    }
}
```

ProducerConsumerExample 是主线程，它启动生产者线程和消费者线程。

```
public class ProducerConsumerExample {
    public static void main(String[] args) {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

不可变对象（Immutable Objects）

如果一个对象它被构造后其，状态不能改变，则这个对象被认为是不可变的（immutable ）。不可变对象的好处是可以创建简单的、可靠的代码。

不可变对象在并发应用种特别有用。因为他们不能改变状态，它们不能被线程干扰所中断或者被其他线程观察到内部不一致的状态。

程序员往往不愿使用不可变对象，因为他们担心创建一个新的对象要比更新对象的成本要高。实际上这种开销常常被过分高估，而且使用不可变对象所带来的一些效率提升也抵消了这种开销。例如：使用不可变对象降低了垃圾回收所产生的额外开销，也减少了用来确保使用可变对象不出现并发错误的一些额外代码。

接下来看一个可变对象的类，然后转化为一个不可变对象的类。通过这个例子说明转化的原则以及使用不可变对象的好处。

一个同步类的例子

SynchronizedRGB 是表示颜色的类，每一个对象代表一种颜色，使用三个整形数表示颜色的三基色，字符串表示颜色名称。

```
public class SynchronizedRGB {
    // Values must be between 0 and 255.
    private int red;
    private int green;
    private int blue;
    private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedRGB(int red,
                           int green,
                           int blue,
                           String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
```

```
        this.name = name;
    }

    public void set(int red,
                    int green,
                    int blue,
                    String name) {
        check(red, green, blue);
        synchronized (this) {
            this.red = red;
            this.green = green;
            this.blue = blue;
            this.name = name;
        }
    }

    public synchronized int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName() {
        return name;
    }

    public synchronized void invert() {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

使用 SynchronizedRGB 时需要小心，避免其处于不一致的状态。例如一个线程执行了以下代码：

```
SynchronizedRGB color =
    new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

如果有另外一个线程在 Statement 1 之后、Statement 2 之前调用了 color.set 方法，那么 myColorInt 的值和 myColorName 的值就会不匹配。为了避免出现这样的结果，必须要像下面这样把这两条语句绑定到一块执行：

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

这种不一致的问题只可能发生在可变对象上。

定义不可变对象的策略

以下的一些创建不可变对象的简单策略。并非所有不可变类都完全遵守这些规则，不过这不是编写这些类的程序员们粗心大意造成的，很可能的是他们有充分的理由确保这些对象在创建后不会被修改。但这需要非常复杂细致的分析，并不适用于初学者。

- 不要提供 setter 方法。（包括修改字段的方法和修改字段引用对象的方法）
- 将类的所有字段定义为 final、private 的。
- 不允许子类重写方法。简单的办法是将类声明为 final，更好的方法是将构造函数声明为私有的，通过工厂方法创建对象。
- 如果类的字段是对可变对象的引用，不允许修改被引用对象。
  - 不提供修改可变对象的方法。
  - 不共享可变对象的引用。当一个引用被当做参数传递给构造函数，而这个引用指向的是一个外部的可变对象时，一定不要保存这个引用。如果必须要保存，那么创建可变对象的拷贝，然后保存拷贝对象的引用。同样如果需要返回内部的可变对象时，不要返回可变对象本身，而是返回其拷贝。

将这一策略应用到 SynchronizedRGB 有以下几步：

- SynchronizedRGB 类有两个 setter 方法。第一个 set 方法只是简单的为字段设值，第二个 invert 方法修改为创建一个新对象，而不是在原有对象上修改。
- 所有的字段都已经是私有的，加上 final 即可。
- 将类声明为 final 的
- 只有一个字段是对象引用，并且被引用的对象也是不可变对象。

经过以上这些修改后，我们得到了 ImmutableRGB：

```
public class ImmutableRGB {
      // Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int red,
                       int green,
                       int blue) {
        if (red < 0 || red > 255
            || green < 0 || green > 255
            || blue < 0 || blue > 255) {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int red,
                        int green,
                        int blue,
                        String name) {
        check(red, green, blue);
        this.red = red;
        this.green = green;
        this.blue = blue;
        this.name = name;
    }


    public int getRGB() {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName() {
        return name;
    }

    public ImmutableRGB invert() {
        return new ImmutableRGB(255 - red,
                       255 - green,
                       255 - blue,
                       "Inverse of " + name);
    }
}
```

高级并发对象

目前为止，之前的教程都是重点讲述了最初作为 Java 平台一部分的低级别 API。这些API 对于非常基本的任务来说已经足够，但是对于更高级的任务就需要更高级的 API。特别是针对充分利用了当今多处理器和多核系统的大规模并发应用程序。 本章，我们将着眼于 Java 5.0 新增的一些高级并发特征。大多数功能已经在新的java.util.concurrent 包中实现。Java 集合框架中也定义了新的并发数据结构。

锁对象

提供了可以简化许多并发应用的锁的惯用法。

同步代码依赖于一种简单的可重入锁。这种锁使用简单，但也有诸多限制。java.util.concurrent.locks 包提供了更复杂的锁。这里会重点关注其最基本的接口 Lock。Lock 对象作用非常类似同步代码使用的内部锁。如同内部锁，每次只有一个线程可以获得 Lock 对象。通过关联 Condition 对象，Lock 对象也支持 wait/notify 机制。

Lock 对象之于隐式锁最大的优势在于，它们有能力收回获得锁的尝试。如果当前锁对象不可用，或者锁请求超时（如果超时时间已指定），tryLock 方法会收回获取锁的请求。如果在锁获取前，另一个线程发送了一个中断，lockInterruptibly 方法也会收回获取锁的请求。

让我们使用 Lock 对象来解决我们在活跃度中见到的死锁问题。Alphonse 和 Gaston 已经把自己训练成能注意到朋友何时要鞠躬。我们通过要求 Friend 对象在双方鞠躬前必须先获得锁来模拟这次改善。下面是改善后模型的源代码 Safelock :

```
public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public boolean impendingBow(Friend bower) {
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                yourLock = bower.lock.tryLock();
```

```
            } finally {
                if (!(myLock && yourLock)) {
                    if (myLock) {
                        lock.unlock();
                    }
                    if (yourLock) {
                        bower.lock.unlock();
                    }
                }
            }
            return myLock && yourLock;
        }

        public void bow(Friend bower) {
            if (impendingBow(bower)) {
                try {
                    System.out.format("%s: %s has" + " bowed to me!%n", this.name, bow
er.getName());
                    bower.bowBack(this);
                } finally {
                    lock.unlock();
                    bower.lock.unlock();
                }
            } else {
                System.out.format(
                        "%s: %s started" + " to bow to me, but saw that" + " I was alr
eady bowing to" + " him.%n",
                        this.name, bower.getName());
            }
        }

        public void bowBack(Friend bower) {
            System.out.format("%s: %s has" + " bowed back to me!%n", this.name, bower.
getName());
        }
    }

    static class BowLoop implements Runnable {
        private Friend bower;
        private Friend bowee;

        public BowLoop(Friend bower, Friend bowee) {
            this.bower = bower;
            this.bowee = bowee;
        }

        public void run() {
            Random random = new Random();
            for (;;) {
                try {
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {
                }
```

```
                bowee.bow(bower);
            }
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new BowLoop(alphonse, gaston)).start();
        new Thread(new BowLoop(gaston, alphonse)).start();
    }
}
```

执行器（Executors）

为加载和管理线程定义了高级 API。Executors 的实现由 java.util.concurrent 包提供，提供了适合大规模应用的线程池管理。

在之前所有的例子中，Thread 对象表示的线程和 Runnable 对象表示的线程所执行的任务之间是紧耦合的。这对于小型应用程序来说没问题，但对于大规模并发应用来说，合理的做法是将线程的创建与管理和程序的其他部分分离开。封装这些功能的对象就是执行器，接下来的部分将讲详细描述执行器。

执行器接口

在 java.util.concurrent 中包括三个执行器接口：

- Executor，一个运行新任务的简单接口。
- ExecutorService，扩展了 Executor 接口。添加了一些用来管理执行器生命周期和任务生命周期的方法。
- ScheduledExecutorService，扩展了 ExecutorService。支持 future 和（或）定期执行任务。

通常来说，指向 executor 对象的变量应被声明为以上三种接口之一，而不是具体的实现类

Executor 接口

Executor 接口只有一个 execute 方法，用来替代通常创建（启动）线程的方法。例如：r 是一个 Runnable 对象，e 是一个 Executor 对象。可以使用

```
e.execute(r);
```

代替

```
(new Thread(r)).start();
```

但 execute 方法没有定义具体的实现方式。对于不同的 Executor 实现，execute 方法可能是创建一个新线程并立即启动，但更有可能是使用已有的工作线程运行r，或者将 r放入到队列中等待可用的工作线程。（我们将在线程池一节中描述工作线程。）

ExecutorService 接口

ExecutorService 接口在提供了 execute 方法的同时，新加了更加通用的 submit 方法。submit 方法除了和 execute 方法一样可以接受 Runnable 对象作为参数，还可以接受 Callable 对象作为参数。使用 Callable对象可以能使任务返还执行的结果。通过 submit 方法返回的 Future 对象可以读取 Callable 任务的执行结果，或是管理 Callable 任务和 Runnable 任务的状态。 ExecutorService 也提供了批量运行 Callable 任务的方法。最后，ExecutorService 还提供了一些关闭执行器的方法。如果需要支持即时关闭，执行器所执行的任务需要正确处理中断。

ScheduledExecutorService 接口

ScheduledExecutorService 扩展 ExecutorService接口并添加了 schedule 方法。调用 schedule 方法可以在指定的延时后执行一个Runnable 或者 Callable 任务。ScheduledExecutorService 接口还定义了按照指定时间间隔定期执行任务的 scheduleAtFixedRate 方法和 scheduleWithFixedDelay 方法。

线程池

线程池是最常见的一种执行器的实现。

在 java.util.concurrent 包中多数的执行器实现都使用了由工作线程组成的线程池，工作线程独立于所它所执行的 Runnable 任务和 Callable 任务，并且常用来执行多个任务。

使用工作线程可以使创建线程的开销最小化。在大规模并发应用中，创建大量的 Thread 对象会占用占用大量系统内存，分配和回收这些对象会产生很大的开销。

一种最常见的线程池是固定大小的线程池。这种线程池始终有一定数量的线程在运行，如果一个线程由于某种原因终止运行了，线程池会自动创建一个新的线程来代替它。需要执行的任务通过一个内部队列提交给线程，当没有更多的工作线程可以用来执行任务时，队列保存额外的任务。

使用固定大小的线程池一个很重要的好处是可以实现优雅退化(degrade gracefully)。例如一个 Web 服务器，每一个 HTTP 请求都是由一个单独的线程来处理的，如果为每一个 HTTP 都创建一个新线程，那么当系统的开销超出其能力时，会突然地对所有请求都停止响应。如果限制 Web 服务器可以创建的线程数量，那么它就不必立即处理所有收到的请求，而是在有能力处理请求时才处理。

创建一个使用线程池的执行器最简单的方法是调用 java.util.concurrent.Executors 的 newFixedThreadPool 方法。Executors 类还提供了下列一下方法：

- newCachedThreadPool 方法创建了一个可扩展的线程池。适合用来启动很多短任务的应用程序。
- newSingleThreadExecutor 方法创建了每次执行一个任务的执行器。
- 还有一些 ScheduledExecutorService 执行器创建的工厂方法。

如果上面的方法都不满足需要，可以尝试 java.util.concurrent.ThreadPoolExecutor 或者java.util.concurrent.ScheduledThreadPoolExecutor。

Fork/Join

该框架是 JDK 7 中引入的并发框架。

fork/join 框架是 ExecutorService 接口的一种具体实现，目的是为了帮助你更好地利用多处理器带来的好处。它是为那些能够被递归地拆解成子任务的工作类型量身设计的。其目的在于能够使用所有可用的运算能力来提升你的应用的性能。

类似于 ExecutorService 接口的其他实现，fork/join 框架会将任务分发给线程池中的工作线程。fork/join 框架的独特之处在与它使用工作窃取(work-stealing)算法。完成自己的工作而处于空闲的工作线程能够从其他仍然处于忙碌(busy)状态的工作线程处窃取等待执行的任务。

fork/join 框架的核心是 ForkJoinPool 类，它是对 AbstractExecutorService 类的扩展。ForkJoinPool 实现了工作窃取算法，并可以执行 ForkJoinTask 任务。

基本使用方法

使用 fork/join 框架的第一步是编写执行一部分工作的代码。你的代码结构看起来应该与下面所示的伪代码类似：

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

翻译为中文为：

```
if (当前这个任务工作量足够小)
    直接完成这个任务
else
    将这个任务或这部分工作分解成两个部分
    分别触发(invoke)这两个子任务的执行，并等待结果
```

你需要将这段代码包裹在一个 ForkJoinTask 的子类中。不过，通常情况下会使用一种更为具体的的类型，或者是 RecursiveTask(会返回一个结果)，或者是 RecursiveAction。 当你的 ForkJoinTask 子类准备好了，创建一个代表所有需要完成工作的对象，然后将其作为参数传递给一个ForkJoinPool 实例的 invoke() 方法即可。

模糊图片的例子

想要了解 fork/join 框架的基本工作原理，接下来的这个例子会有所帮助。假设你想要模糊一张图片。原始的 source 图片由一个整数的数组表示，每个整数表示一个像素点的颜色数值。与 source 图片相同，模糊之后的 destination 图片也由一个整数数组表示。 对图片的模糊操作是通过对 source 数组中的每一个像素点进行处理完成的。处理的过程是这样的：将每个像素点的色值取出，与周围像素的色值（红、黄、蓝三个组成部分）放在一起取平均值，得到的结果被放入 destination 数组。因为一张图片会由一个很大的数组来表示，这个流程会花费一段较长的时间。如果使用 fork/join 框架来实现这个模糊算法，你就能够借助多处理器系统的并行处理能力。下面是上述算法结合 fork/join 框架的一种简单实现：

```
public class ForkBlur extends RecursiveAction {
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;

    // Processing window size; should be odd.
    private int mBlurWidth = 15;

    public ForkBlur(int[] src, int start, int length, int[] dst) {
        mSource = src;
        mStart = start;
        mLength = length;
        mDestination = dst;
    }

    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
            // Calculate average.
            float rt = 0, gt = 0, bt = 0;
            for (int mi = -sidePixels; mi <= sidePixels; mi++) {
                int mindex = Math.min(Math.max(mi + index, 0),
                                    mSource.length - 1);
                int pixel = mSource[mindex];
                rt += (float)((pixel & 0x00ff0000) >> 16)
                      / mBlurWidth;
                gt += (float)((pixel & 0x0000ff00) >>  8)
                      / mBlurWidth;
                bt += (float)((pixel & 0x000000ff) >>  0)
                      / mBlurWidth;
            }

            // Reassemble destination pixel.
            int dpixel = (0xff000000     ) |
                   (((int)rt) << 16) |
                   (((int)gt) <<  8) |
                   (((int)bt) <<  0);
            mDestination[index] = dpixel;
        }
    }

  ...
```

接下来你需要实现父类中的 compute() 方法，它会直接执行模糊处理，或者将当前的工作拆分成两个更小的任务。数组的长度可以作为一个简单的阀值来判断任务是应该直接完成还是应该被拆分。

```
protected static int sThreshold = 100000;

protected void compute() {
    if (mLength < sThreshold) {
        computeDirectly();
        return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
              new ForkBlur(mSource, mStart + split, mLength - split,
                           mDestination));
}
```

如果前面这个方法是在一个 RecursiveAction 的子类中，那么设置任务在ForkJoinPool 中执行就再直观不过了。通常会包含以下一些步骤：

1. 创建一个表示所有需要完成工作的任务。

   // source image pixels are in src // destination image pixels are in dst ForkBlur fb = new ForkBlur(src, 0, src.length, dst);

2. 创建将要用来执行任务的 ForkJoinPool。

   ForkJoinPool pool = new ForkJoinPool();

3. 执行任务。

   pool.invoke(fb);

想要浏览完成的源代码，请查看 ForkBlur示例，其中还包含一些创建 destination 图片文件的额外代码。

标准实现

除了能够使用 fork/join 框架来实现能够在多处理系统中被并行执行的定制化算法（如前文中的 ForkBlur.java 例子），在 Java SE 中一些比较常用的功能点也已经使用 fork/join 框架来实现了。在 Java SE 8 中，java.util.Arrays 类的一系列parallelSort() 方法就使用了 fork/join 来实现。这些方法与 sort() 方法很类似，但是通过使用 fork/join框 架，借助了并发来完成相关工作。在多处理器系统中，对大数组的并行排序会比串行排序更快。这些方法究竟是如何运用 fork/join 框架并不在本教程的讨论范围内。想要了解更多的信息，请参见 Java API 文档。其他采用了 fork/join 框架的方法还包括java.util.streams包中的一些方法，此包是作为 Java SE 8 发行版中 Project Lambda 的一部分。想要了解更多信息，请参见 Lambda 表达式一节。

并发集合

并发集合简化了大型数据集合管理，且极大的减少了同步的需求。

java.util.concurrent 包囊括了 Java 集合框架的一些附加类。它们也最容易按照集合类所提供的接口来进行分类：

- BlockingQueue 定义了一个先进先出的数据结构，当你尝试往满队列中添加元素，或者从空队列中获取元素时，将会阻塞或者超时。
- ConcurrentMap 是 java.util.Map 的子接口，定义了一些有用的原子操作。移除或者替换键值对的操作只有当 key 存在时才能进行，而新增操作只有当 key 不存在时。使这些操作原子化，可以避免同步。ConcurrentMap 的标准实现是 ConcurrentHashMap，它是 HashMap 的并发模式。
- ConcurrentNavigableMap 是 ConcurrentMap 的子接口，支持近似匹配。ConcurrentNavigableMap 的标准实现是 ConcurrentSkipListMap，它是 TreeMap 的并发模式。

所有这些集合，通过在集合里新增对象和访问或移除对象的操作之间，定义一个happens-before 的关系，来帮助程序员避免内存一致性错误。

原子变量

java.util.concurrent.atomic 包定义了对单一变量进行原子操作的类。所有的类都提供了 get 和 set 方法，可以使用它们像读写 volatile 变量一样读写原子类。就是说，同一变量上的一个 set 操作对于任意后续的 get 操作存在 happens-before 关系。原子的 compareAndSet 方法也有内存一致性特点，就像应用到整型原子变量中的简单原子算法。

为了看看这个包如何使用，让我们返回到最初用于演示线程干扰的 Counter 类：

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

使用同步是一种使 Counter 类变得线程安全的方法，如 SynchronizedCounter：

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

对于这个简单的类，同步是一种可接受的解决方案。但是对于更复杂的类，我们可能想要避免不必要同步所带来的活跃度影响。将 int 替换为 AtomicInteger 允许我们在不进行同步的情况下阻止线程干扰，如 AtomicCounter：

```
import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }
}
```

并发随机数

并发随机数（JDK7）提供了高效的多线程生成伪随机数的方法。

在 JDK7 中，java.util.concurrent 包含了一个相当便利的类 ThreadLocalRandom，可以在当应用程序期望在多个线程或 ForkJoinTasks 中使用随机数时使用。

对于并发访问，使用 TheadLocalRandom 代替 Math.random() 可以减少竞争，从而获得更好的性能。

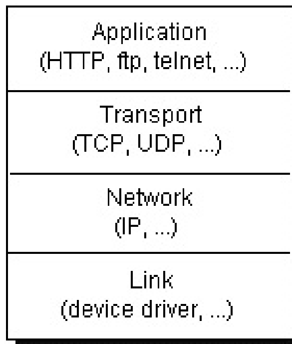
你只需调用 ThreadLocalRandom.current()， 然后调用它的其中一个方法去获取一个随机数即可。下面是一个例子：

```
int r = ThreadLocalRandom.current().nextInt(4, 77);
```

集合框架

网络基础

在互联网上之间的通信交流，一般是基于 TCP (Transmission Control Protocol，传输控制协议) 或者 UDP (User Datagram Protocol，用户数据报协议) ，如下图：



编写 Java 应用，我们只需关注于应用层 （application layer），而不用关心 TCP 和 UDP 所在的传输层是如何实现的。java.net 包含了你编程所需的类，这些类是与操作系统无关的。比如 URL, URLConnection, Socket, 和 ServerSocket 类是使用 TCP 连接网络的，DatagramPacket, DatagramSocket, 和 MulticastSocket 类是用于 UDP 的。

Java 支持的协议只有 TCP 和 UDP ，以及在建立在 TCP 和 UDP 之上其他应用层协议。所有其他传输层、网际层和更底层的协议，如 ICMP、IGMP、ARP、RARP、RSVP 和其他协议在 Java 中只能链接到原生代码来实现。

TCP

TCP (Transmission Control Protocol) 是面向连接的、提供端到端可靠的数据流(flow of data)。TCP 提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。

“面向连接”就是在正式通信前必须要与对方建立起连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机说“喂”，然后才说明是谁。

三次握手

TCP 是基于连接的协议，也就是说，在正式收发数据前，必须和对方建立可靠的连接。一个 TCP 连接必须要经过三次“握手”才能建立起来，简单的讲就是：

1. 主机 A 向主机 B 发出连接请求数据包：“我想给你发数据，可以吗？”；
2. 主机 B 向主机 A 发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你来吧”；
3. 主机 A 再发出一个数据包确认主机 B 的要求同步：“好的，我来也，你接着吧！”

三次“握手”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机 A 才向主机 B 正式发送数据。

可以详见《TCP 协议的三次握手、四次分手》

如何保证数据的可靠

TCP 通过下列方式来提供可靠性：

- 应用数据被分割成 TCP 认为最适合发送的数据块。这和 UDP 完全不同，应用程序产生的数据报长度将保持不变。由 TCP 传递给 IP 的信息单位称为报文段或段（segment）。
- 当 TCP 发出一个段后，它启动一个定时器，等待目的端确认收到这个报文段。如果不能及时收到一个确认，将重发这个报文段。（可自行了解 TCP 协议中自适应的超时及重传策略）。
- 当 TCP 收到发自 TCP 连接另一端的数据，它将发送一个确认。这个确认不是立即发送，通常将推迟几分之一秒。
- TCP 将保持它首部和数据的检验和。这是一个端到端的检验和，目的是检测数据在传输过程中的任何变化。如果收到段的检验和有差错， TCP 将丢弃这个报文段和不确认收到此报文段（希望发送端超时并重发）。
- 既然 TCP 报文段作为 IP 数据报来传输，而 IP 数据报的到达可能会失序，因此 TCP 报文段的到达也可能会失序。如果必要， TCP 将对收到的数据进行重新排序，将收到的数据以正确的顺序交给应用层。
- 既然 IP 数据报会发生重复， TCP 的接收端必须丢弃重复的数据。
- TCP 还能提供流量控制。TCP 连接的每一方都有固定大小的缓冲空间。 TCP 的接收端只允许另一端发送接收端缓冲区所能接纳的数据。这将防止较快主机致使较慢主机的缓冲区溢出。

UDP

UDP (User Datagram Protocol) 不是面向连接的，主机发送独立的数据报（datagram）给其他主机，不保证数据到达。由于 UDP 在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。

而无连接是一开始就发送信息（严格说来，这是没有开始、结束的），只是一次性的传递，是先不需要接受方的响应，因而在一定程度上也无法保证信息传递的可靠性了，就像写信一样，我们只是将信寄出去，却不能保证收信人一定可以收到。

TCP 和 UDP 如何抉择

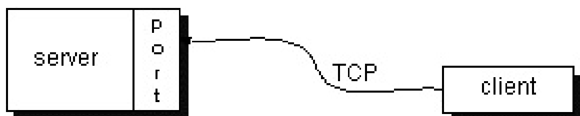
TCP 是面向连接的，有比较高的可靠性， 一些要求比较高的服务一般使用这个协议，如 FTP、Telnet、SMTP、HTTP、POP3 等，而 UDP 是面向无连接的，使用这个协议的常见服务有 DNS、SNMP、QQ 等。对于 QQ 必须另外说明一下，QQ2003 以前是只使用UDP协议的，其服务器使用8000端口，侦听是否有信息传来，客户端使用 4000 端口，向外发送信息（这也就不难理解在一般的显IP的QQ版本中显示好友的IP地址信息中端口常为4000或其后续端口的原因了），即QQ程序既接受服务又提供服务，在以后的 QQ 版本中也支持使用 TCP 协议了。

端口

一般来说，一台计算机具有单个物理连接到网络。数据通过这个连接去往特定的计算机。然而，该数据可以被用于在计算机上运行的不同应用。那么，计算机知道哪个应用程序转发数据？通过使用端口。

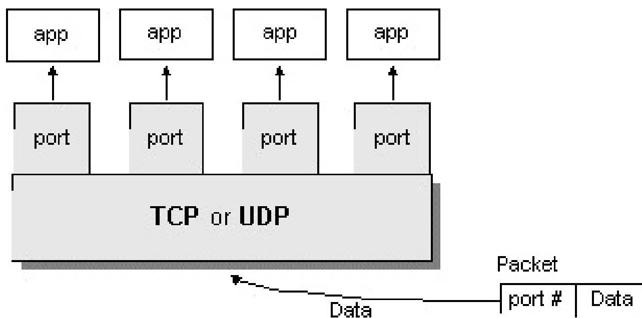
在互联网上传输的数据是通过计算机的标识和端口来定位的。计算机的标识是 32-bit 的 IP 地址。端口由一个 16-bit 的数字。

在诸如面向连接的通信如 TCP，服务器应用将套接字绑定到一个特定端口号。这是向系统注册服务用来接受该端口的数据。然后，客户端可以在与服务器在服务器端口会合，如下图所示：



TCP 和 UDP 协议使用的端口来将接收到的数据映射到一个计算机上运行的进程。

在基于数据报的通信，如 UDP，数据报包中包含它的目的地的端口号，然后 UDP 将数据包路由到相应的应用程序，如本图所示的端口号：



端口号取值范围是从 0 到 65535 （16-bit 长度），其中范围从 0 到 1023 是受限的，它们是被知名的服务所保留使用，例如 HTTP （端口是 80）和 FTP （端口是20、21）等系统服务。这些端口被称为众所周知的端口（well-known ports）。您的应用程序不应该试图绑定到他们。你可以访问 http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml 来查询各种常用的已经分配的端口号列表。

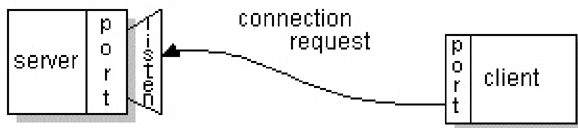
Socket

什么是 Socket

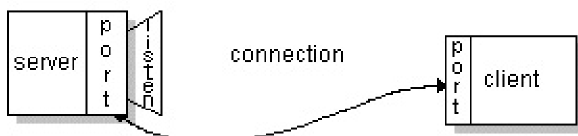
Socket（套接字）：是在网络上运行两个程序之间的双向通信链路的一个端点。socket绑定到一个端口号，使得 TCP 层可以标识数据最终要被发送到哪个应用程序。

正常情况下，一台服务器在特定计算机上运行，并具有被绑定到特定端口号的 socket。服务器只是等待，并监听用于客户发起的连接请求的 socket 。

在客户端：客户端知道服务器所运行的主机名称以及服务器正在侦听的端口号。建立连接请求时，客户端尝试与主机服务器和端口会合。客户端也需要在连接中将自己绑定到本地端口以便于给服务器做识别。本地端口号通常是由系统分配的。



如果一切顺利的话，服务器接受连接。一旦接受，服务器获取绑定到相同的本地端口的新 socket ，并且还具有其远程端点设定为客户端的地址和端口。它需要一个新的socket，以便它可以继续监听原来用于客户端连接请求的 socket 。



在客户端，如果连接被接受，则成功地创建一个套接字和客户端可以使用该 socket 与服务器进行通信。

客户机和服务器现在可以通过 socket 写入或读取来交互了。

端点是IP地址和端口号的组合。每个 TCP 连接可以通过它的两个端点被唯一标识。这样，你的主机和服务器之间可以有多个连接。

java.net 包中提供了一个类 Socket，实现您的 Java 程序和网络上的其他程序之间的双向连接。 Socket 类隐藏任何特定系统的细节。通过使用 java.net.Socket 类，而不是依赖于原生代码，Java 程序可以用独立于平台的方式与网络进行通信。

此外，java.net 包含了 ServerSocket 类，它实现了服务器的 socket 可以侦监听和接受客户端的连接。下文将展示如何使用 Socket 和 ServerSocket 类。

实现一个 echo 服务器

让我们来看看这个例子，程序可以建立使用 Socket 类连接到服务器程序，客户端可以通过 socket 向服务器发送数据和接收数据。

EchoClient 示例程序实现了一个客户端，连接到回声服务器。回声服务器从它的客户端接收数据并原样返回回来。EchoServer 实现了 echo 服务器。（客户端可以连接到支持 Echo 协议的任何主机）

EchoClient 创建一个 socket，从而得到回声服务器的连接。它从标准输入流中读取用户输入，然后通过 socket 转发该文本给回声服务器。服务器通过该 socket 将文本原样输入回给客户端。客户机程序读取并显示从服务器传递回给它的数据。

注意，EchoClient 例子既从 socket 写入又从 socket 中读取数据。

EchoClient 代码：

```
public class EchoClient {
    public static void main(String[] args) throws IOException {

        if (args.length != 2) {
            System.err.println(
                "Usage: java EchoClient <host name> <port number>");
            System.exit(1);
        }

        String hostName = args[0];
        int portNumber = Integer.parseInt(args[1]);

        try (
            Socket echoSocket = new Socket(hostName, portNumber);
            PrintWriter out =
                new PrintWriter(echoSocket.getOutputStream(), true);
            BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(echoSocket.getInputStream()));
            BufferedReader stdIn =
                new BufferedReader(
                    new InputStreamReader(System.in))
        ) {
            String userInput;
            while ((userInput = stdIn.readLine()) != null) {
                out.println(userInput);
                System.out.println("echo: " + in.readLine());
            }
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host " + hostName);
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection to " +
                hostName);
            System.exit(1);
        }
    }
}
```

EchoServer 代码：

```
public class EchoServer {
    public static void main(String[] args) throws IOException {

        if (args.length != 1) {
            System.err.println("Usage: java EchoServer <port number>");
            System.exit(1);
        }

        int portNumber = Integer.parseInt(args[0]);

        try (
            ServerSocket serverSocket =
                new ServerSocket(Integer.parseInt(args[0]));
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);

            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + portNumber + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

首先启动服务器，在命令行输入如下，设定一个端口号，比如 7（Echo 协议指定端口是 7）：

```
java EchoServer 7
```

而后启动客户端，echoserver.example.com 是你主机的名称，如果是本机的话，主机名称可以是 localhost

```
java EchoClient echoserver.example.com 7
```

输出效果如下：

```
你好吗？
echo: 你好吗？
我很好哦
echo: 我很好哦
要过年了，www.waylau.com 祝你 猴年大吉，身体健康哦！
echo: 要过年了，www.waylau.com 祝你 猴年大吉，身体健康哦！
```

I/O 模型的演进

什么是同步？什么是异步？阻塞和非阻塞又有什么区别？本文先从 Unix 的 I/O 模型讲起，介绍了5种常见的 I/O 模型。而后再引出 Java 的 I/O 模型的演进过程，并用实例说明如何选择合适的 Java I/O 模型来提高系统的并发量和可用性。

由于，Java 的 I/O 依赖于操作系统的实现，所以先了解 Unix 的 I/O 模型有助于理解 Java 的 I/O。

相关概念

同步和异步

描述的是用户线程与内核的交互方式：

- 同步是指用户线程发起 I/O 请求后需要等待或者轮询内核 I/O 操作完成后才能继续执行；
- 异步是指用户线程发起 I/O 请求后仍继续执行，当内核 I/O 操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

阻塞和非阻塞

描述的是用户线程调用内核 I/O 操作的方式：

- 阻塞是指 I/O 操作需要彻底完成后才返回到用户空间；
- 非阻塞是指 I/O 操作被调用后立即返回给用户一个状态值，无需等到 I/O 操作彻底完成。

一个 I/O 操作其实分成了两个步骤：发起 I/O 请求和实际的 I/O 操作。 阻塞 I/O 和非阻塞 I/O 的区别在于第一步，发起 I/O 请求是否会被阻塞，如果阻塞直到完成那么就是传统的阻塞 I/O ，如果不阻塞，那么就是非阻塞 I/O 。 同步 I/O 和异步 I/O 的区别就在于第二个步骤是否阻塞，如果实际的 I/O 读写阻塞请求进程，那么就是同步 I/O 。

Unix I/O 模型

Unix 下共有五种 I/O 模型：

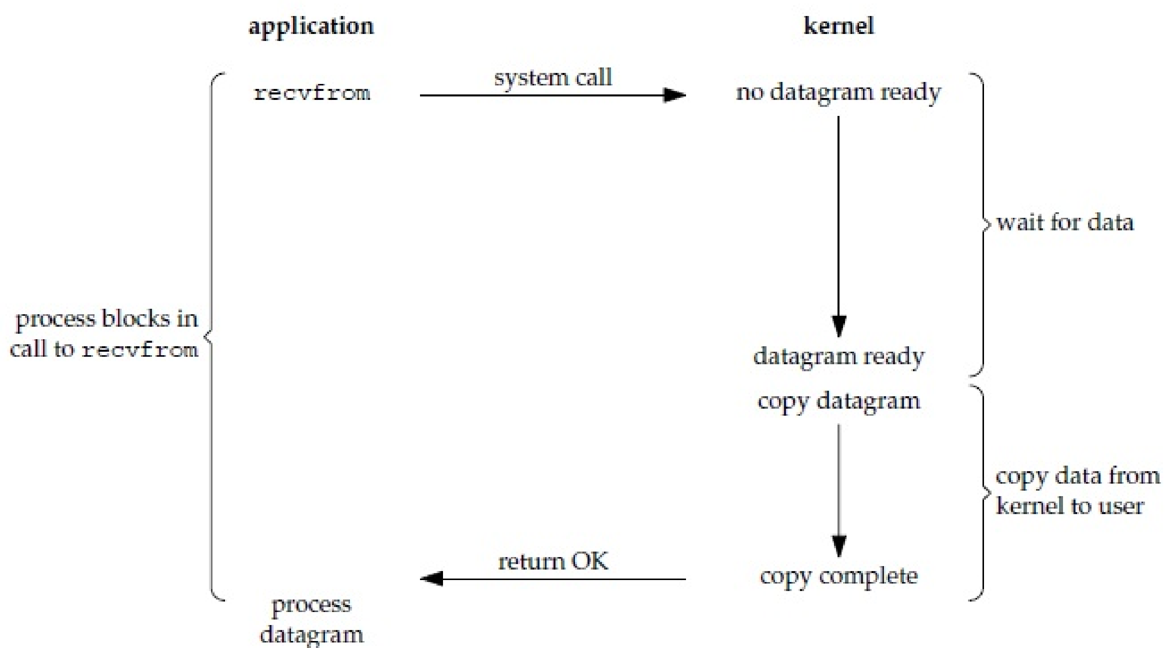
1. 阻塞 I/O
2. 非阻塞 I/O
3. I/O 复用（select 和 poll）
4. 信号驱动 I/O（SIGIO）
5. 异步 I/O（Posix.1 的 aio_ 系列函数）

注：若读者想深入了解 Unix 的网络知识，推荐阅读《Unix Network Programming》，文本只简单介绍下这五种模型，文中的图例也引用自该书的图例。

阻塞 I/O

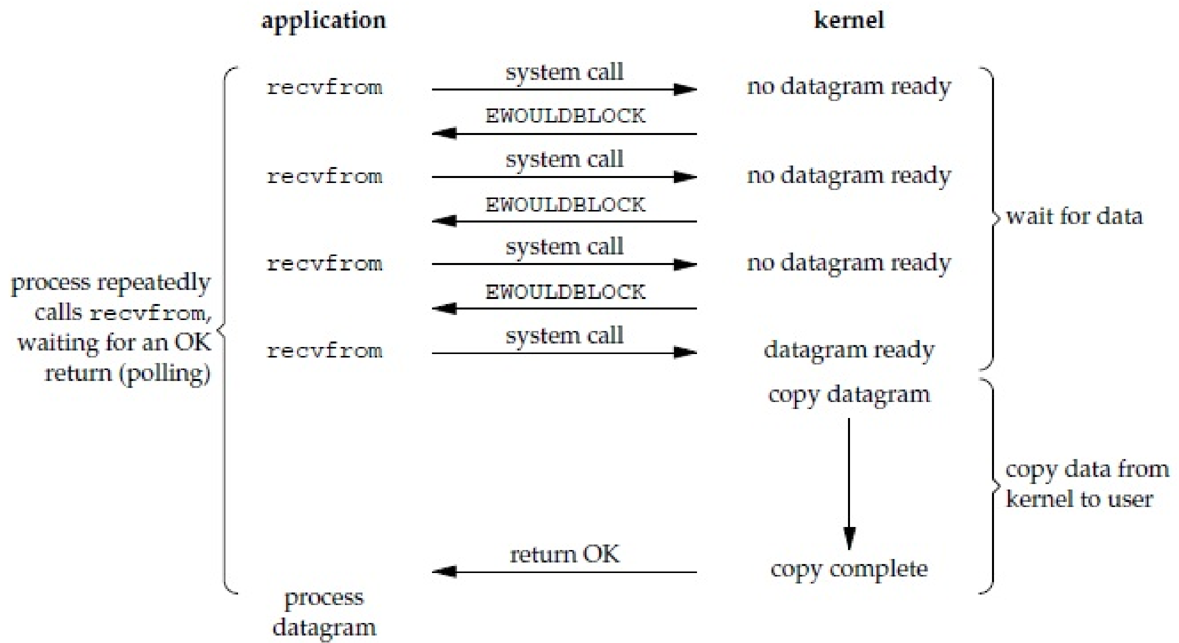
请求无法立即完成则保持阻塞。

- 阶段1：等待数据就绪。网络 I/O 的情况就是等待远端数据陆续抵达；磁盘I/O的情况就是等待磁盘数据从磁盘上读取到内核态内存中。
- 阶段2：数据拷贝。出于系统安全,用户态的程序没有权限直接读取内核态内存,因此内核负责把内核态内存中的数据拷贝一份到用户态内存中。



非阻塞 I/O

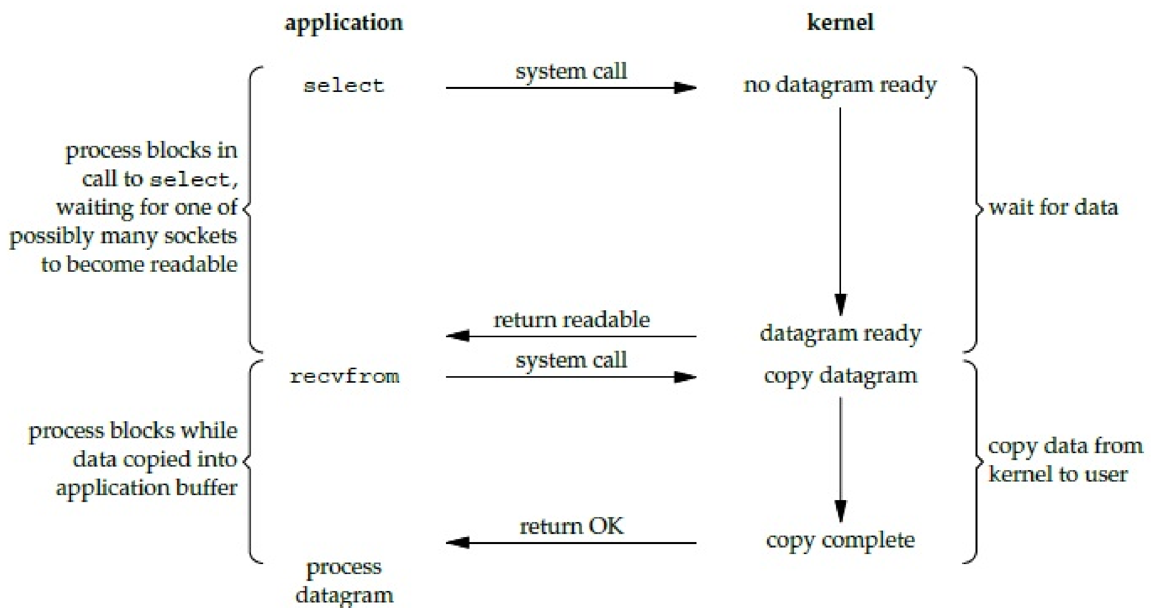
- socket 设置为 NONBLOCK（非阻塞）就是告诉内核，当所请求的 I/O 操作无法完成时，不要将进程睡眠，而是返回一个错误码(EWOULDBLOCK)，这样请求就不会阻塞
- I/O 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续测试，直到数据准备好为止。整个 I/O 请求的过程中，虽然用户线程每次发起 I/O 请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的 CPU 的资源
- 数据准备好了，从内核拷贝到用户空间。



一般很少直接使用这种模型，而是在其他 I/O 模型中使用非阻塞 I/O 这一特性。这种方式对单个 I/O 请求意义不大,但给 I/O 多路复用铺平了道路.

I/O 复用（异步阻塞 I/O）

I/O 复用会用到 select 或者 poll 函数，这两个函数也会使进程阻塞，但是和阻塞 I/O 所不同的的，这两个函数可以同时阻塞多个 I/O 操作。而且可以同时对多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。



从流程上来看，使用 select 函数进行 I/O 请求和同步阻塞模型没有太大的区别，甚至还多了添加监视 socket，以及调用 select 函数的额外操作，效率更差。但是，使用 select 以后最大的优势是用户可以在一个线程内同时处理多个 socket 的 I/O 请求。用户可以注册多个 socket，然后不断地调用 select 读取被激活的 socket，即可达到在同一个线程内同时处理多个 I/O 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

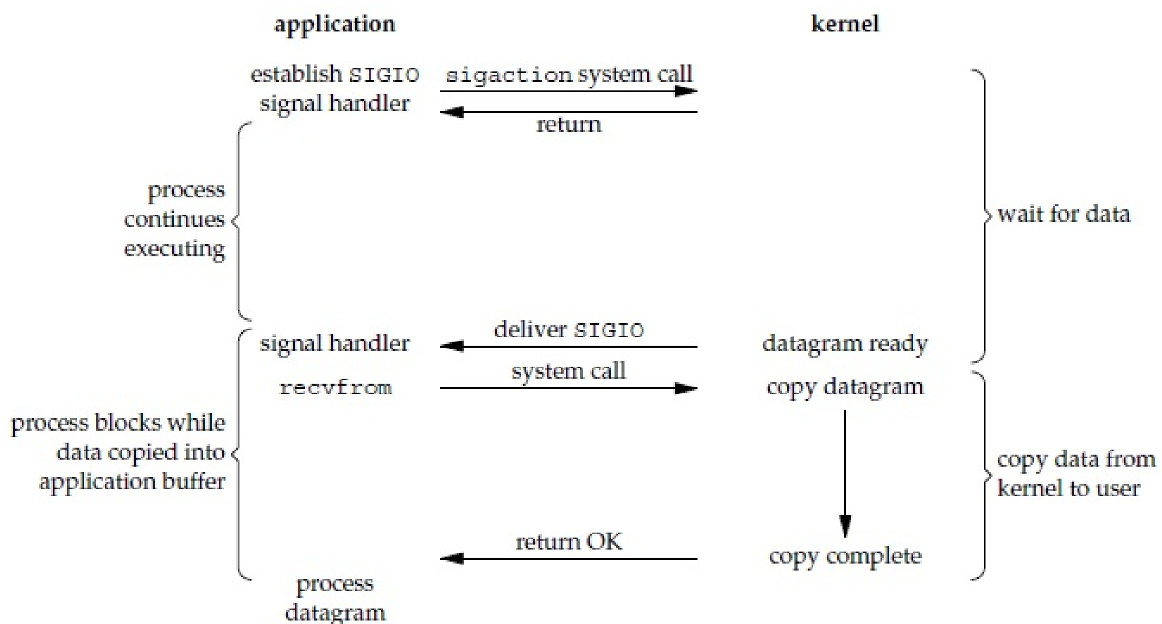
I/O 多路复用模型使用了 Reactor 设计模式实现了这一机制。

注：有关“Reactor 设计模式”请可参阅 https://en.wikipedia.org/wiki/Reactor_pattern。

调用 select / poll 该方法由一个用户态线程负责轮询多个 socket,直到某个阶段1的数据就绪,再通知实际的用户线程执行阶段2的拷贝。 通过一个专职的用户态线程执行非阻塞 I/O 轮询,模拟实现了阶段1的异步化。

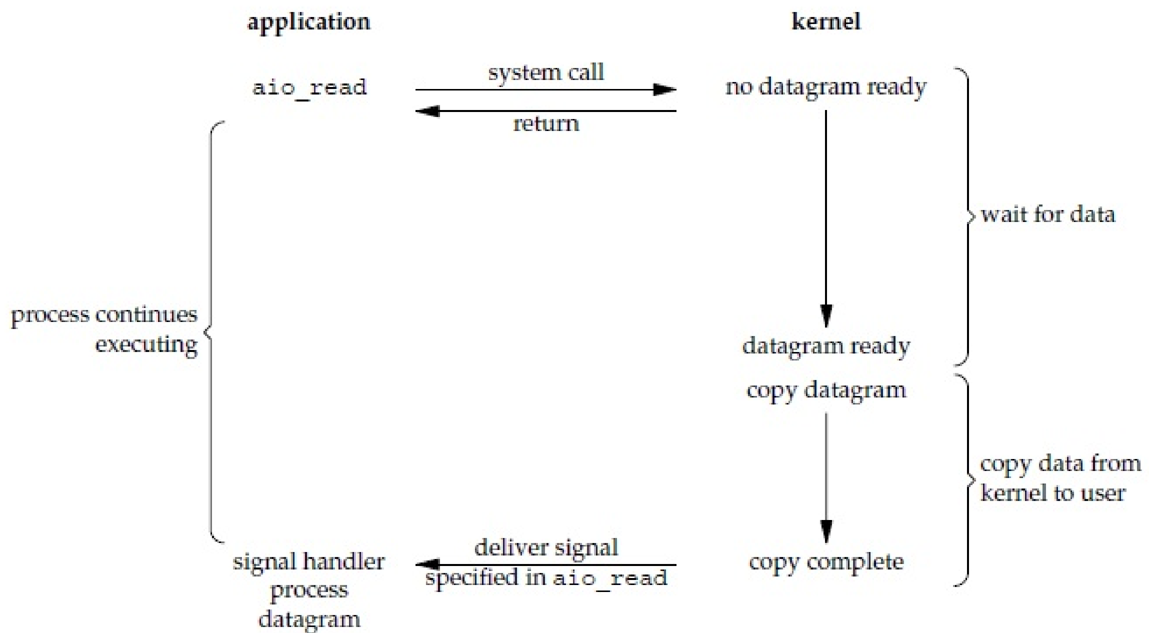
信号驱动 I/O（SIGIO）

首先我们允许 socket 进行信号驱动 I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 SIGIO 信号，可以在信号处理函数中调用 I/O 操作函数处理数据。



异步 I/O

调用 aio_read 函数，告诉内核描述字，缓冲区指针，缓冲区大小，文件偏移以及通知的方式，然后立即返回。当内核将数据拷贝到缓冲区后，再通知应用程序。



异步 I/O 模型使用了 Proactor 设计模式实现了这一机制。

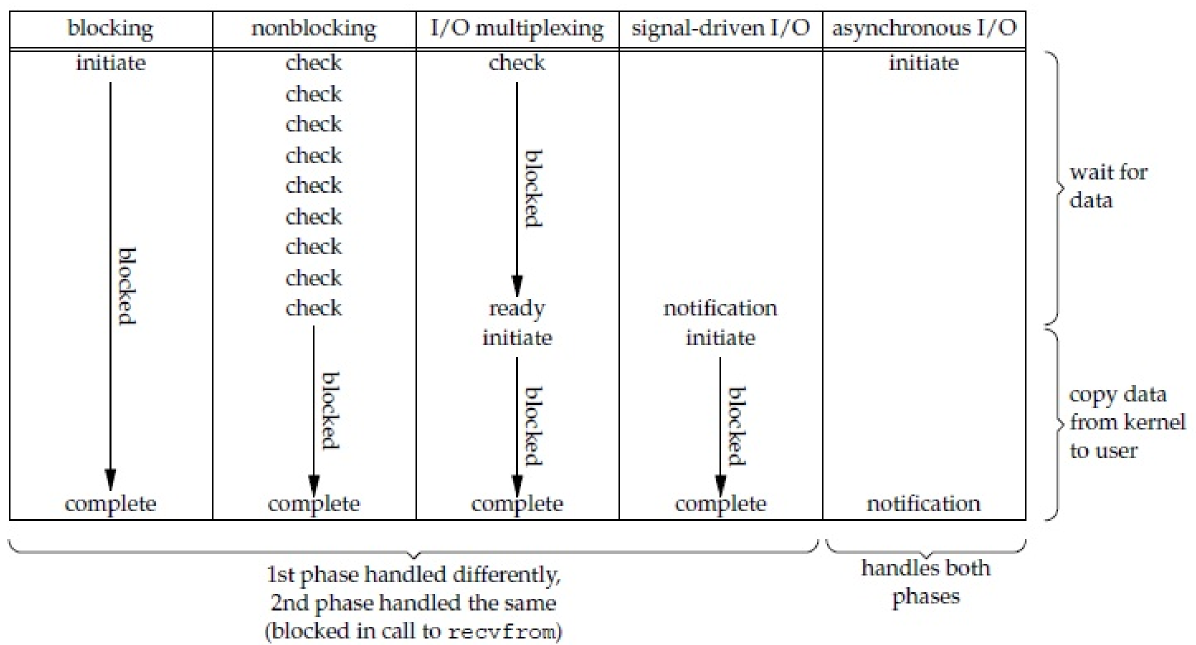
注：有关“Proactor 设计模式”可以参阅 https://en.wikipedia.org/wiki/Proactor_pattern。

告知内核,当整个过程(包括阶段1和阶段2)全部完成时,通知应用程序来读数据.

几种 I/O 模型的比较

前四种模型的区别是阶段1不相同，阶段2基本相同，都是将数据从内核拷贝到调用者的缓冲区。而异步 I/O 的两个阶段都不同于前四个模型。

同步 I/O 操作引起请求进程阻塞，直到 I/O 操作完成。异步 I/O 操作不引起请求进程阻塞。



常见 Java I/O 模型

在了解了 UNIX 的 I/O 模型之后，其实 Java 的 I/O 模型也是类似。

“阻塞I/O”模式

在上一节 Socket 章节中的 EchoServer 就是一个简单的阻塞 I/O 例子，服务器启动后，等待客户端连接。在客户端连接服务器后，服务器就阻塞读写取数据流。

EchoServer 代码：

```java
public class EchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        try (
            ServerSocket serverSocket =
                new ServerSocket(port);
            Socket clientSocket = serverSocket.accept();
            PrintWriter out =
                new PrintWriter(clientSocket.getOutputStream(), true);

            BufferedReader in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        ) {
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println("Exception caught when trying to listen on port "
                + port + " or listening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

改进为“阻塞I/O+多线程”模式

使用多线程来支持多个客户端来访问服务器。

主线程 MultiThreadEchoServer.java

```
public class MultiThreadEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        Socket clientSocket = null;
        try (ServerSocket serverSocket = new ServerSocket(port);) {
            while (true) {
                clientSocket = serverSocket.accept();

                // MultiThread
                new Thread(new EchoServerHandler(clientSocket)).start();
            }
        } catch (IOException e) {
            System.out.println(
                    "Exception caught when trying to listen on port " + port + " or li
stening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

处理器类 EchoServerHandler.java

```
public class EchoServerHandler implements Runnable {
    private Socket clientSocket;

    public EchoServerHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
    public void run() {
        try (PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
                BufferedReader in = new BufferedReader(new InputStreamReader(clientSoc
ket.getInputStream()));) {

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                out.println(inputLine);
            }
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

存在问题：每次接收到新的连接都要新建一个线程，处理完成后销毁线程，代价大。当有大量地短连接出现时，性能比较低。

改进为“阻塞I/O+线程池”模式

针对上面多线程的模型中，出现的线程重复创建、销毁带来的开销，可以采用线程池来优化。每次接收到新连接后从池中取一个空闲线程进行处理，处理完成后再放回池中，重用线程避免了频率地创建和销毁线程带来的开销。

主线程 ThreadPoolEchoServer.java

```
public class ThreadPoolEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        ExecutorService threadPool = Executors.newFixedThreadPool(5);
        Socket clientSocket = null;
        try (ServerSocket serverSocket = new ServerSocket(port);) {
            while (true) {
                clientSocket = serverSocket.accept();

                // Thread Pool
                threadPool.submit(new Thread(new EchoServerHandler(clientSocket)));
            }
        } catch (IOException e) {
            System.out.println(
                    "Exception caught when trying to listen on port " + port + " or li
stening for a connection");
            System.out.println(e.getMessage());
        }
    }
}
```

存在问题：在大量短连接的场景中性能会有提升，因为不用每次都创建和销毁线程，而是重用连接池中的线程。但在大量长连接的场景中，因为线程被连接长期占用，不需要频繁地创建和销毁线程，因而没有什么优势。

虽然这种方法可以适用于小到中度规模的客户端的并发数，如果连接数超过 100,000或更多，那么性能将很不理想。

改进为“非阻塞I/O”模式

"阻塞I/O+线程池"网络模型虽然比"阻塞I/O+多线程"网络模型在性能方面有提升，但这两种模型都存在一个共同的问题：读和写操作都是同步阻塞的,面对大并发（持续大量连接同时请求）的场景，需要消耗大量的线程来维持连接。CPU 在大量的线程之间频繁切换，性能损耗很大。一旦单机的连接超过1万，甚至达到几万的时候，服务器的性能会急剧下降。

而 NIO 的 Selector 却很好地解决了这个问题，用主线程（一个线程或者是 CPU 个数的线程）保持住所有的连接，管理和读取客户端连接的数据，将读取的数据交给后面的线程池处理，线程池处理完业务逻辑后，将结果交给主线程发送响应给客户端，少量的线程就可以处理大量连接的请求。

Java NIO 由以下几个核心部分组成：

- Channel
- Buffer
- Selector

要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

主线程 NonBlokingEchoServer.java

```
public class NonBlokingEchoServer {
    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {

        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }
        System.out.println("Listening for connections on port " + port);

        ServerSocketChannel serverChannel;
        Selector selector;
        try {
            serverChannel = ServerSocketChannel.open();
            InetSocketAddress address = new InetSocketAddress(port);
            serverChannel.bind(address);
            serverChannel.configureBlocking(false);
            selector = Selector.open();
            serverChannel.register(selector, SelectionKey.OP_ACCEPT);
        } catch (IOException ex) {
            ex.printStackTrace();
            return;
        }

        while (true) {
            try {
                selector.select();
            } catch (IOException ex) {
                ex.printStackTrace();
```

```
                    break;
                }
                Set<SelectionKey> readyKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = readyKeys.iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    iterator.remove();
                    try {
                        if (key.isAcceptable()) {
                            ServerSocketChannel server = (ServerSocketChannel) key.channel
();

                            SocketChannel client = server.accept();
                            System.out.println("Accepted connection from " + client);
                            client.configureBlocking(false);
                            SelectionKey clientKey = client.register(selector,
                                    SelectionKey.OP_WRITE | SelectionKey.OP_READ);
                            ByteBuffer buffer = ByteBuffer.allocate(100);
                            clientKey.attach(buffer);
                        }
                        if (key.isReadable()) {
                            SocketChannel client = (SocketChannel) key.channel();
                            ByteBuffer output = (ByteBuffer) key.attachment();
                            client.read(output);
                        }
                        if (key.isWritable()) {
                            SocketChannel client = (SocketChannel) key.channel();
                            ByteBuffer output = (ByteBuffer) key.attachment();
                            output.flip();
                            client.write(output);

                            output.compact();
                        }
                    } catch (IOException ex) {
                        key.cancel();
                        try {
                            key.channel().close();
                        } catch (IOException cex) {
                        }
                    }
                }
            }

        }
}
```

改进为“异步I/O”模式

Java SE 7 版本之后，引入了异步 I/O （NIO.2） 的支持，为构建高性能的网络应用提供了一个利器。

主线程 AsyncEchoServer.java

```
public class AsyncEchoServer {

    public static int DEFAULT_PORT = 7;

    public static void main(String[] args) throws IOException {
        int port;

        try {
            port = Integer.parseInt(args[0]);
        } catch (RuntimeException ex) {
            port = DEFAULT_PORT;
        }

        ExecutorService taskExecutor = Executors.newCachedThreadPool(Executors.default
ThreadFactory());
        // create asynchronous server socket channel bound to the default group
        try (AsynchronousServerSocketChannel asynchronousServerSocketChannel = Asynchr
onousServerSocketChannel.open()) {
            if (asynchronousServerSocketChannel.isOpen()) {
                // set some options
                asynchronousServerSocketChannel.setOption(StandardSocketOptions.SO_RCV
BUF, 4 * 1024);
                asynchronousServerSocketChannel.setOption(StandardSocketOptions.SO_REU
SEADDR, true);
                // bind the server socket channel to local address
                asynchronousServerSocketChannel.bind(new InetSocketAddress(port));
                // display a waiting message while ... waiting clients
                System.out.println("Waiting for connections ...");
                while (true) {
                    Future<AsynchronousSocketChannel> asynchronousSocketChannelFuture
= asynchronousServerSocketChannel
                            .accept();
                    try {
                        final AsynchronousSocketChannel asynchronousSocketChannel = as
ynchronousSocketChannelFuture
                                .get();
                        Callable<String> worker = new Callable<String>() {
                            @Override
                            public String call() throws Exception {
                                String host = asynchronousSocketChannel.getRemoteAddre
ss().toString();

                                System.out.println("Incoming connection from: " + host
);

                                final ByteBuffer buffer = ByteBuffer.allocateDirect(10
24);

                                // transmitting data
                                while (asynchronousSocketChannel.read(buffer).get() !=
 -1) {

                                    buffer.flip();
                                    asynchronousSocketChannel.write(buffer).get();
```

```
                                    if (buffer.hasRemaining()) {
                                        buffer.compact();
                                    } else {
                                        buffer.clear();
                                    }
                                }
                                asynchronousSocketChannel.close();
                                System.out.println(host + " was successfully served!")
;
                                return host;
                            }
                        };
                        taskExecutor.submit(worker);
                    } catch (InterruptedException | ExecutionException ex) {
                        System.err.println(ex);
                        System.err.println("\n Server is shutting down ...");
                        // this will make the executor accept no new threads
                        // and finish all existing threads in the queue
                        taskExecutor.shutdown();
                        // wait until all threads are finished
                        while (!taskExecutor.isTerminated()) {
                        }
                        break;
                    }
                }
            } else {
                System.out.println("The asynchronous server-socket channel cannot be o
pened!");
            }
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

源码

本章例子的源码，可以在 https://github.com/waylau/essential-java 中 `com.waylau.essentialjava.net.echo` 包下找到。

JDBC

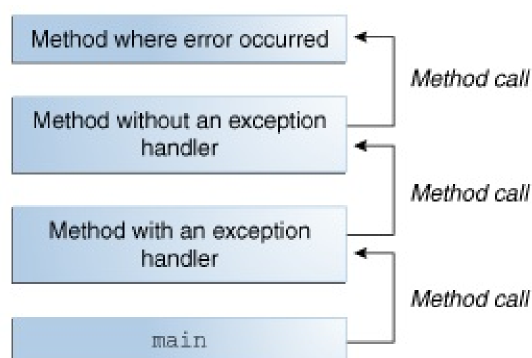
异常

在Java语言中，是使用“异常（exception）”来处理错误及其他异常事件。术语“异常”是短语“异常事件（exceptional event）”的缩写。

异常是在程序执行期间发生的事件，它会中断程序指令的正常流程。

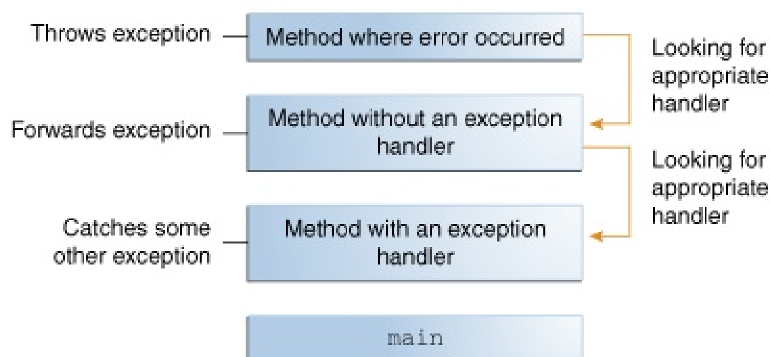
当在方法中发生错误时，该方法创建一个对象并将其移交给运行时系统。 该对象称为“异常对象（exception object）”，包含有关错误的信息，包括错误发生时其类型和程序的状态。 创建异常对象并将其移交给运行时系统，这个过程被称为“抛出异常（throwing an exception）”。

在方法抛出异常后，运行时系统会尝试寻找一些方式来处理它。 这个方法列表被叫做“调用堆栈（call　stack）”，调用方式如下图所示（参见下图）。



运行时系统搜寻包含能够处理异常的代码块的方法所请求的堆栈。这个代码块叫做“异常处理器（ exception handler）”，搜寻首先从发生的方法开始，然后依次按着调用方法的倒序检索调用堆栈。当找到一个相应的处理器时，运行时系统就把异常传递给这个处理器。一个异常处理器要适当地考虑抛出的异常对象的类型与异常处理器所处理的异常的类型是否匹配。

当异常处理器被选中时，称为“捕获异常（catch the exception）”。异常被捕获以后，异常处理器关闭。如果运行时系统搜寻了这个方法的所有调用堆栈，而没有找到相应的异常处理器，如下图所示，运行进系统将终止执行。



使用异常来管理错误比传统的错误管理技术有一些优势。见“使用异常带来的优势”一节。

异常捕获与处理

本节介绍如何使用三个异常处理程序组件（try、catch 和 finally）来编写异常处理程序。 然后，介绍了 Java SE 7中引入的 try-with-resources 语句。 try-with-resources 语句特别适合于使用 `Closeable` 的资源（例如流）的情况。

本节的最后一部分将通过一个示例来分析在各种情况下发生的情况。

以下示例定义并实现了一个名为ListOfNumbers的类。 构造时，ListOfNumbers 创建一个ArrayList，其中包含10个序列值为0到9的整数元素。ListOfNumbers类还定义了一个名为writeList的方法，该方法将数列表写入一个名为 `OutFile.txt` 的文本文件中。 此示例使用在 `java.io` 中定义的输出类，这些类包含在基本I/O中。

```
// Note: This class will not compile yet.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
    // The FileWriter constructor throws IOException, which must be caught.
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException, which must be cau
ght.
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

构造函数 FileWriter 初始化文件上的输出流。如果文件无法打开，构造函数会抛出一个IOException异常。第二个对ArrayList类的get方法的调用，如果其参数的值太小（小于0）或太大（超过ArrayList当前包含的元素数量），它将抛出 IndexOutOfBoundsException。

如果尝试编译ListOfNumbers类，则编译器将打印有关FileWriter构造函数抛出的异常的错误消息。但是，它不显示有关get抛出的异常的错误消息。原因是构造函数IOException抛出的异常是一个检查异常，而get方法IndexOutOfBoundsException抛出的异常是未检查的异常。

现在，我们已经熟悉ListOfNumbers类，并且知道了其中那些地方可能抛出异常。下一步我们就可以编写异常处理程序来捕获和处理这些异常。

try块

构造异常处理程序的第一步是封装可能在try块中抛出异常的代码。 一般来说，try块看起来像下面这样：

```
try {
    code
}
catch and finally blocks . . .
```

示例标记 `code` 中的段可以包含一个或多个可能抛出的异常。

每行可能抛出异常的代码都可以用单独的一个 try 块，或者多个异常放置在一个 try 块中。 以下示例由于非常简短，所有使用一个try块。

```
private List<Integer> list;
private static final int SIZE = 10;

public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entered try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    }
    catch and finally blocks  . . .
}
```

如果在try块中发生异常，那么该异常由与其相关联的异常处理程序将会进行处理。 要将异常处理程序与try块关联，必须在其后面放置一个catch块。

catch块

通过在try块之后直接提供一个或多个catch块，可以将异常处理程序与try块关联。 在try块的结尾和第一个catch块的开始之间没有代码。

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

每个catch块是一个异常处理程序，处理由其参数指示的异常类型。 参数类型ExceptionType声明了处理程序可以处理的异常类型，并且必须是从Throwable类继承的类的名称。 处理程序可以使用名称引用异常。

catch块包含了在调用异常处理程序时执行的代码。 当处理程序是调用堆栈中第一个与ExceptionType匹配的异常抛出的类型时，运行时系统将调用异常处理程序。 如果抛出的对象可以合法地分配给异常处理程序的参数，则系统认为它是匹配。

以下是writeList方法的两个异常处理程序：

```
try {

} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

异常处理程序可以做的不仅仅是打印错误消息或停止程序。 它们可以执行错误恢复，提示用户做出决定，或者使用异常链将错误传播到更高级别的处理程序，如“异常链”部分所述。

在一个异常处理程序中处理多个类型的异常

在Java SE 7和更高版本中，单个catch块可以处理多种类型的异常。 此功能可以减少代码重复，并减少定义过于宽泛的异常。

在catch子句中，多个类型的异常使用竖线（|）分隔每个异常类型：

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

注意：如果catch块处理多个异常类型，则catch参数将隐式为final。 在本示例中，catch参数ex是final，因此您不能在catch块中为其分配任何值。

finally 块

finally块总是在try块退出时执行。这确保即使发生意外异常也会执行finally块。但 finally 的用处不仅仅是异常处理 - 它允许程序员避免清理代码意外绕过 return、continue 或 break 。将清理代码放在finally块中总是一个好的做法，即使没有预期的异常。

注意：如果在执行try或catch代码时JVM退出，则finally块可能无法执行。同样，如果执行try或catch代码的线程被中断或杀死，则finally块可能不执行，即使应用程序作为一个整体继续。

writeList方法的try块打开一个PrintWriter。程序应该在退出writeList方法之前关闭该流。这提出了一个有点复杂的问题，因为writeList的try块可以以三种方式中的一种退出。

- new FileWriter语句失败并抛出IOException。
- list.get(i)语句失败，并抛出IndexOutOfBoundsException。
- 一切成功，try块正常退出。

运行时系统总是执行finally块内的语句，而不管try块内发生了什么。所以它是执行清理的完美场所。

下面的finally块为writeList方法清理，然后关闭PrintWriter。

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

重要：finally块是防止资源泄漏的关键工具。 当关闭文件或恢复资源时，将代码放在finally块中，以确保资源始终恢复。

考虑在这些情况下使用try-with-resources语句，当不再需要时自动释放系统资源。

源码

本章例子的源码，可以在 https://github.com/waylau/essential-java 中 `com.waylau.essentialjava.exception` 包下找到。

try-with-resources 语句

try-with-resources 是 JDK 7 中一个新的异常处理机制，它能够很容易地关闭在 try-catch 语句块中使用的资源。所谓的资源（resource）是指在程序完成后，必须关闭的对象。try-with-resources 语句确保了每个资源在语句结束时关闭。所有实现了 java.lang.AutoCloseable 接口（其中，它包括实现了 java.io.Closeable 的所有对象），可以使用作为资源。

例如，我们自定义一个资源类

```
public class Demo {
    public static void main(String[] args) {
        try(Resource res = new Resource()) {
            res.doSome();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class Resource implements AutoCloseable {
    void doSome() {
        System.out.println("do something");
    }
    @Override
    public void close() throws Exception {
        System.out.println("resource is closed");
    }
}
```

执行输出如下：

```
do something
resource is closed
```

可以看到，资源终止被自动关闭了。

再来看一个例子，是同时关闭多个资源的情况：

```
public class Main2 {
    public static void main(String[] args) {
        try(ResourceSome some = new ResourceSome();
             ResourceOther other = new ResourceOther()) {
            some.doSome();
            other.doOther();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

class ResourceSome implements AutoCloseable {
    void doSome() {
        System.out.println("do something");
    }
    @Override
    public void close() throws Exception {
        System.out.println("some resource is closed");
    }
}

class ResourceOther implements AutoCloseable {
    void doOther() {
        System.out.println("do other things");
    }
    @Override
    public void close() throws Exception {
        System.out.println("other resource is closed");
    }
}
```

最终输出为：

```
do something
do other things
other resource is closed
some resource is closed
```

在 try 语句中越是最后使用的资源，越是最早被关闭。

try-with-resources 在 JDK 9 中的改进

作为 Milling Project Coin 的一部分, try-with-resources 声明在 JDK 9 已得到改进。如果你已经有一个资源是 final 或等效于 final 变量,您可以在 try-with-resources 语句中使用该变量，而无需在 try-with-resources 语句中声明一个新变量。

例如,给定资源的声明

```
// A final resource
final Resource resource1 = new Resource("resource1");
// An effectively final resource
Resource resource2 = new Resource("resource2");
```

老方法编写代码来管理这些资源是类似的:

```
// Original try-with-resources statement from JDK 7 or 8
try (Resource r1 = resource1;
     Resource r2 = resource2) {
    // Use of resource1 and resource 2 through r1 and r2.
}
```

而新方法可以是

```
// New and improved try-with-resources statement in JDK 9
try (resource1;
     resource2) {
    // Use of resource1 and resource 2.
}
```

看上去简洁很多吧。对 Java 未来的发展信心满满。

愿意尝试 JDK 9 这种新语言特性的可以下载使用 JDK 9 快照。Enjoy!

源码

本章例子的源码，可以在 https://github.com/waylau/essential-java 中 `com.waylau.essentialjava.exception.trywithresources` 包下找到。

通过方法声明异常抛出

上一节展示了如何为ListOfNumbers类中的writeList方法编写异常处理程序。 有时，它适合代码捕获可能发生在其中的异常。 但在其他情况下，最好让一个方法进一步推给上层来调用堆栈处理异常。 例如，如果您将ListOfNumbers类提供为类包的一部分，则可能无法预期包的所有用户的需求。 在这种情况下，最好不要捕获异常，并允许一个方法进一步推给上层来调用堆栈来处理它。

如果writeList方法没有捕获其中可能发生的已检查异常，则writeList方法必须指定它可以抛出这些异常。 让我们修改原始的writeList方法来指定它可以抛出的异常，而不是捕捉它们。 请注意，下面是不能编译的writeList方法的原始版本。

```
public void writeList() {
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = " + list.get(i));
    }
    out.close();
}
```

要指定writeList可以抛出两个异常，请为writeList方法的方法声明添加一个throws子句。throws子句包含throws关键字，后面是由该方法抛出的所有异常的逗号分隔列表。 该子句在方法名和参数列表之后，在定义方法范围的大括号之前。这里是一个例子。

```
public void writeList() throws IOException, IndexOutOfBoundsException {
```

记住 IndexOutOfBoundsException是未检查异常（unchecked exception），包括它在throws子句中不是强制性的。 你可以写成下面这样

```
public void writeList() throws IOException {
```

如何抛出异常

在你可以捕获异常之前，一些代码必须抛出一个异常。任何代码都可能会抛出异常：您的代码，来自其他人编写的包（例如Java平台附带的包）或Java运行时环境的代码。无论是什么引发的异常，它总是通过 throw 语句抛出。

您可能已经注意到，Java平台提供了许多异常类。所有类都是Throwable类的后代，并且都允许程序区分在程序执行期间可能发生的各种类型的异常。

您还可以创建自己的异常类来表示在您编写的类中可能发生的问题。事实上，如果您是包开发人员，您可能必须创建自己的一组异常类，以允许用户区分包中可能发生的错误与Java平台或其他包中发生的错误。

您还可以创建异常链。有关更多信息，请参阅“异常链”部分。

throw语句

所有方法都使用throw语句抛出异常。 throw语句需要一个参数：throwable 对象。 Throwable对象是Throwable类的任何子类的实例。 这里是一个throw 语句的例子。

```
throw someThrowableObject;
```

让我们来看一下上下文中的throw语句。 以下pop方法取自实现公共堆栈对象的类。 该方法从堆栈中删除顶层元素并返回对象。

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

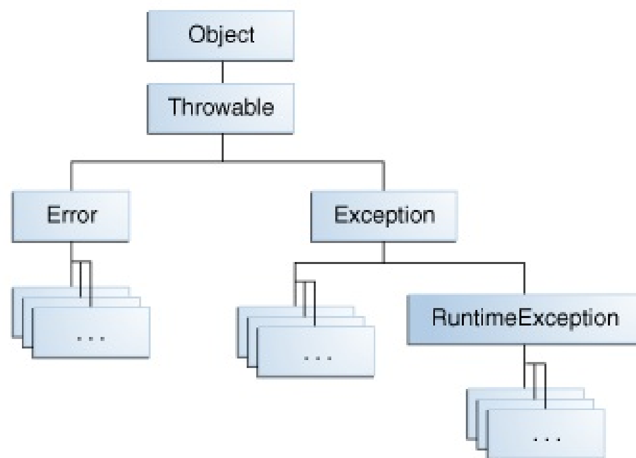
pop 方法将会检查栈中的元素。如果栈是空的（它的size等于0），则pop实例化一个EmptyStackException对象（java.util的成员）并抛出它。 本章中的“创建异常类”部分介绍如何创建自己的异常类。 现在，所有你需要记住的是，你可以只抛出继承自

java.lang.Throwable类的对象。

注意，pop方法的声明不包含throws子句。 EmptyStackException不是已检查异常，因此不需要pop来声明它可能发生。

Throwable类及其子类

继承自Throwable类的对象包括直接后代（直接从Throwable类继承的对象）和间接后代（从Throwable类的子代或孙代继承的对象）。 下图说明了Throwable类及其最重要的子类的类层次结构。 正如你所看到的，Throwable有两个直接的后代：Error和Exception。



Error 类

当Java虚拟机中发生动态链接故障或其他硬故障时，虚拟机会抛出 Error。简单的程序通常不捕获或抛出Error。

Exception 类

大多数程序抛出和捕获从 Exception 类派生的对象。Exception 表示发生了问题，但它不是严重的系统问题。你编写的大多数程序将抛出并捕获Exception而不是 Error。

Java平台定义了 Exception 类的许多后代。这些后代表示可能发生的各种类型的异常。例如，IllegalAccessException表示找不到特定方法，NegativeArraySizeException表示程序尝试创建一个负大小的数组。

一个 Exception 子类RuntimeException保留用于指示不正确使用API的异常。运行时异常的一个示例是NullPointerException，当方法尝试通过空引用访问对象的成员时，会发生此异常。“未检查异常”章节讨论了为什么大多数应用程序不应该抛出运行时异常或RuntimeException的子类。

异常链

应用程序通常会通过抛出另一个异常来响应异常。 实际上，第一个异常引起第二个异常。 它可以是非常有助于用户知道什么时候一个异常导致另一个异常。“异常链（Chained Exceptions）”帮助程序员做到这一点。

以下是Throwable中支持异常链的方法和构造函数。

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

initCause和Throwable构造函数的Throwable参数是导致当前异常的异常。 getCause返回导致当前异常的异常，initCause设置当前异常的原因。

以下示例显示如何使用异常链。

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

在此示例中，当捕获到IOException时，将创建一个新的SampleException异常，并附加原始的异常原因，并将异常链抛出到下一个更高级别的异常处理程序。

访问堆栈跟踪信息

现在让我们假设更高级别的异常处理程序想要以自己的格式转储堆栈跟踪。

定义：堆栈跟踪（stack trace）提供有关当前线程的执行历史的信息，并列出在异常发生时调用的类和方法的名称。 堆栈跟踪是一个有用的调试工具，通常在抛出异常时会利用它。

以下代码显示了如何在异常对象上调用getStackTrace方法。

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

日志 API

如果要记录catch块中所发生异常，最好不要手动解析堆栈跟踪并将输出发送到System.err()，而是使用java.util.logging包中的日志记录工具将输出发送到文件。

```
try {
    Handler handler = new FileHandler("OutFile.log");
    Logger.getLogger("").addHandler(handler);

} catch (IOException e) {
    Logger logger = Logger.getLogger("package.name");
    StackTraceElement elements[] = e.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        logger.log(Level.WARNING, elements[i].getMethodName());
    }
}
```

创建异常类

当面对选择抛出异常的类型时，您可以使用由别人编写的异常 - Java平台提供了许多可以使用的异常类 - 或者您可以编写自己的异常类。 如果您对任何以下问题回答“是”，您应该编写自己的异常类；否则，你可以使用别人的。

- 你需要一个Java平台中没有表示的异常类型吗？
- 如果用户能够区分你的异常与由其他供应商编写的类抛出的异常吗？
- 你的代码是否抛出不止一个相关的异常？
- 如果您使用他人的例外，用户是否可以访问这些异常？ 一个类似的问题是你的包是独立只提供自己使用吗？

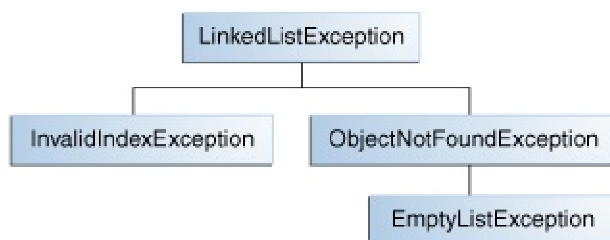
一个例子

假设你正在写一个链表类。该类支持以下方法：

- objectAt(int n) - 返回列表中第n个位置的对象。如果参数小于0或大于当前列表中的对象数，则抛出异常。
- firstObject() - 返回列表中的第一个对象。如果列表不包含对象，则抛出异常。
- indexOf(Object o) - 搜索指定对象的列表，并返回其在列表中的位置。如果传入方法的对象不在列表中，则抛出异常。

链表类可以抛出多个异常，使用一个异常处理程序捕获链表所抛出的所有异常是很方便的。此外，如果您计划在包中分发链表，所有相关代码都应打包在一起。因此，链表应该提供自己的一组异常类。

下图说明了链表抛出的异常的一个可能的类层次结构。



选择超类

任何 Exception 子类都可以用作 LinkedListException 的父类。 然而，但这些子类有些专用的，有些又与 LinkedListException 完全无关。 因此，LinkedListException的父类应该是Exception。

你编写的大多数applet和应用程序都会抛出 Exception 对象。 Error 通常用于系统中严重的硬错误，例如阻止JVM运行的错误。

注意：对于可读代码，最好将字符串Exception附加到从异常类继承（直接或间接）的所有类的名称。

未检查异常

因为Java编程语言不需要捕获方法或声明未检查异常（包括 RuntimeException、Error及其子类），程序员可能会试图编写只抛出未检查异常的代码，或使所有异常子类继承自RuntimeException。这两个快捷方式都允许程序员编写代码，而不必担心编译器错误，也不用担心声明或捕获任何异常。虽然这对于程序员似乎很方便，但它避开了捕获或者声明异常的需求，并且可能会导致其他人在使用您的类而产生问题。

为什么设计人员决定强制一个方法来指定所有可以抛出的未捕获的已检查异常？任何可以由方法抛出的 Exception 都是方法的公共编程接口的一部分。调用方法的人必须知道一个方法可以抛出的异常，以便他们可以决定如何处理它们。这些异常是该方法的编程接口的一部分，作为它的参数和 return 值。

下一个问题可能是："既然一个方法的API已经做好了很好的记录，包括它可以抛出的异常，为什么不指定运行时异常？"运行时异常展示的是编程问题的结果，因此，API用户可能会用不合理方式来处理它们。这样就有可能产生问题，包括算术异常，例如除以零;指针异常，例如试图通过空引用访问对象;索引异常，例如尝试通过太大或太小的索引访问数组元素。

运行时异常可能发生在程序中的任何地方，在典型的程序中它们可以非常多。必须在每个方法声明中添加运行时异常则会降低程序的清晰度。因此，编译器不需要捕获或声明运行时异常（尽管可以是可以做到）。

一种情况是，通常的做法是当用户调用一个方法不正确时，抛出一个RuntimeException。例如，一个方法可以检查其中一个参数是否不正确为null。如果参数为null，那么该方法可能会抛出NullPointerException异常，这是一个未检查异常。

一般来说，不要抛出一个RuntimeException或创建一个RuntimeException的子类，这样你就不会被声明哪些方法可以抛出的异常所困扰。

一个底线原则是：如果客户端可以合理地从期望异常中恢复，那么使其成为一个已检查异常。如果客户端无法从异常中恢复，请将其设置为未检查异常。

使用异常带来的优势

现在你知道什么是异常，以及如何使用它们，现在是时候了解在程序中使用异常的优点。

优点1：将错误处理代码与“常规”代码分离

异常提供了一种方法来分离当一个程序的主逻辑发生异常情况时应该做什么的细节。 在传统的编程中，错误检测、报告和处理常常导致混淆意大利面条代码（spaghetti code）。 例如，考虑这里的伪代码方法将整个文件读入内存。

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

乍一看，这个功能看起来很简单，但它忽略了以下所有潜在错误。

- 如果无法打开文件会发生什么？
- 如果无法确定文件的长度，会发生什么？
- 如果不能分配足够的内存，会发生什么？
- 如果读取失败会发生什么？
- 如果文件无法关闭会怎么样？

为了处理这种情况，readFile函数必须有更多的代码来执行错误检测\报告和处理。 这里是一个示例，来展示该函数可能会是什么样子。

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

这里面会有很多错误检测、报告的细节，使得原来的七行代码被淹没在这杂乱的代码中。 更糟的是，代码的逻辑流也已经丢失，因此很难判断代码是否正确：如果函数无法分配足够的内存，文件是否真的被关闭？ 在编写方法三个月后修改方法时，更难以确保代码能够继续正确的操作。 因此，许多程序员通过简单地忽略它来解决这个问题。这样当他们的程序崩溃时，就生成了报告错误。

异常使您能够编写代码的主要流程，并处理其他地方的特殊情况。 如果readFile函数使用异常而不是传统的错误管理技术，它将看起来更像下面。

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

请注意，异常不会减少你在法执行检测、报告和处理错误方面的工作，但它们可以帮助您更有效地组织工作。

优点2：将错误沿调用推栈向上传递

异常的第二个优点是能够在方法的调用堆栈上将错误向上传递。 假设 readFile 方法是由主程序进行的一系列嵌套方法调用中的第四个方法：method1调用method2，它调用了method3，最后调用readFile。

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

还假设method1是对readFile中可能发生的错误感兴趣的唯一方法。 传统的错误通知技术强制method2和method3将readFile返回的错误代码传递到调用堆栈，直到错误代码最终到达method1 - 对它们感兴趣的唯一方法。

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

回想一下，Java运行时环境通过调用堆栈向后搜索以找到任何对处理特定异常感兴趣的方法。 一个方法可以阻止在其中抛出的任何异常，从而允许一个方法在调用栈上更远的地方来捕获它。 因此，只有关心错误的方法才需要担心检测错误。

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

然而，如伪代码所示，抛弃异常需要中间人方法的一些努力。 任何可以在方法中抛出的已检查异常都必须在其throws子句中指定。

优点3：对错误类型进行分组和区分

因为在程序中抛出的所有异常都是对象，异常的分组或分类是类层次结构的自然结果。 Java平台中一组相关异常类的示例是java.io - IOException中定义的那些异常类及其后代。IOException是最常见的，表示执行I/O时可能发生的任何类型的错误。 它的后代表示更具体的错误。 例如，FileNotFoundException意味着文件无法在磁盘上找到。

一个方法可以编写可以处理非常特定异常的特定处理程序。 FileNotFoundException类没有后代，因此下面的处理程序只能处理一种类型的异常。

```
catch (FileNotFoundException e) {
    ...
}
```

方法可以通过在catch语句中指定任何异常的超类来基于其组或常规类型捕获异常。 例如，为了捕获所有I/O异常，无论其具体类型如何，异常处理程序都会指定一个IOException参数。

```
catch (IOException e) {
    ...
}
```

这个处理程序将能够捕获所有I/O异常，包括FileNotFoundException、EOFException等等。您可以通过查询传递给异常处理程序的参数来查找有关发生的详细信息。 例如，使用以下命令打印堆栈跟踪。

```
catch (IOException e) {
    // Output goes to System.err.
    e.printStackTrace();
    // Send trace to stdout.
    e.printStackTrace(System.out);
}
```

下面例子可以处理所有的异常：

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

Exception 类接近Throwable类层次结构的顶部。因此，这个处理程序将会捕获除处理程序想要捕获的那些异常之外的许多其他异常。在程序中如果是以这种方式来处理异常，那么你程序一般的做法就是，例如，是打印出一个错误消息给用户，然后退出。

在大多数情况下，异常处理程序应该尽可能的具体。原因是处理程序必须做的第一件事是在选择最佳恢复策略之前，首先要确定发生的是什么类型的异常。实际上，如果不捕获特定的错误，处理程序必须适应任何可能性。太过通用的异常处理程序可能会捕获和处理程序员不期望的并且处理程序不想要的异常，从而使代码更容易出错。

如上所述，您可以以常规方式创建异常分组来处理异常，也可以使用特定的异常类型来区分异常从而可以以确切的方式来处理异常。

附录

参考引用

- Core Java Tenth Edition
- Thinking in Java Fourth Edition
- The Well-Grounded Java Developer
- The Java Tutorial, Sixth Edition
- TCP/IP Illustrated Volume 1: The Protocols
- Java Network Programming, 4th Edition
- Pro Java 7 NIO.2
- Unix Network Programming, Volume 1: The Sockets Networking API (3rd Edition)